
The Python Library Reference

Version 3.6.11

**Guido van Rossum
and the Python development team**

juin 29, 2020

**Python Software Foundation
Email : docs@python.org**

Table des matières

1	Introduction	3
2	Fonctions natives	5
3	Constantes natives	25
3.1	Constantes ajoutées par le module <code>site</code>	26
4	Types natifs	27
4.1	Valeurs booléennes	27
4.2	Opérations booléennes — <code>and</code> , <code>or</code> , <code>not</code>	28
4.3	Comparaisons	28
4.4	Types numériques — <code>int</code> , <code>float</code> , <code>complex</code>	29
4.5	Les types itérateurs	34
4.6	Types séquentiels — <code>list</code> , <code>tuple</code> , <code>range</code>	35
4.7	Type Séquence de Texte — <code>str</code>	41
4.8	Séquences Binaires — <code>bytes</code> , <code>bytearray</code> , <code>memoryview</code>	51
4.9	Types d'ensembles — <code>set</code> , <code>frozenset</code>	70
4.10	Les types de correspondances — <code>dict</code>	73
4.11	Le type gestionnaire de contexte	76
4.12	Autres types natifs	77
4.13	Attributs spéciaux	79
5	Exceptions natives	81
5.1	Classes de base	82
5.2	Exceptions concrètes	83
5.3	Avertissements	88
5.4	Hierarchie des exceptions	89
6	Services de Manipulation de Texte	91
6.1	<code>string</code> — Opérations usuelles sur des chaînes	91
6.2	<code>re</code> — Opérations à base d'expressions rationnelles	102
6.3	<code>difflib</code> — Helpers for computing deltas	122
6.4	<code>textwrap</code> — Encapsulation et remplissage de texte	132
6.5	<code>unicodedata</code> — Base de données Unicode	136
6.6	<code>stringprep</code> — Préparation des chaînes de caractères internet	138
6.7	<code>readline</code> — interface pour GNU <i>readline</i>	139
6.8	<code>rlcompleter</code> — Fonction de complétion pour GNU <i>readline</i>	144

7	Services autour des Données Binaires	145
7.1	struct — manipulation de données agrégées sous forme binaire comme une séquence d'octets	145
7.2	codecs — Registre des codecs et classes de base associées	151
8	Types de données	169
8.1	datetime — Types de base pour la date et l'heure	169
8.2	calendar — General calendar-related functions	199
8.3	collections — Types de données de conteneurs	202
8.4	collections.abc — Classes de base abstraites pour les conteneurs	218
8.5	heapq — File de priorité basée sur un tas	222
8.6	bisect — Algorithme de bisection de listes	226
8.7	array — Tableaux efficaces de valeurs numériques	228
8.8	weakref — Weak references	231
8.9	types — Dynamic type creation and names for built-in types	239
8.10	copy — Opérations de copie superficielle et récursive	242
8.11	pprint — L'affichage élégant de données	243
8.12	reprlib — Alternate repr() implementation	249
8.13	enum — Énumérations	251
9	Modules numériques et mathématiques	269
9.1	numbers — Classes de base abstraites numériques	269
9.2	Fonctions mathématiques — math	272
9.3	Fonctions mathématiques pour nombres complexes — cmath	278
9.4	decimal — Arithmétique décimale en virgule fixe et flottante	281
9.5	fractions — Nombres rationnels	306
9.6	random — Génère des nombres pseudo-aléatoires	308
9.7	statistics — Mathematical statistics functions	315
10	Modules de programmation fonctionnelle	321
10.1	itertools — Fonctions créant des itérateurs pour boucler efficacement	321
10.2	functools — Fonctions de haut niveau et opérations sur des objets appelables	336
10.3	operator — Opérateurs standards en tant que fonctions	342
11	Accès aux Fichiers et aux Dossiers	351
11.1	pathlib — Chemins de système de fichiers orientés objet	351
11.2	os.path — manipulation courante des chemins	367
11.3	fileinput — Iterate over lines from multiple input streams	372
11.4	stat — Interpreting stat() results	374
11.5	filecmp — Comparaisons de fichiers et de répertoires	379
11.6	tempfile — Génération de fichiers et répertoires temporaires	381
11.7	glob — Recherche de chemins de style Unix selon certains motifs	385
11.8	fnmatch — Filtrage par motif des noms de fichiers Unix	386
11.9	linecache — Accès direct aux lignes d'un texte	387
11.10	shutil — Opérations de haut niveau sur les fichiers	388
11.11	macpath — Fonctions de manipulation de chemins pour Mac OS 9	395
12	Persistance des données	397
12.1	pickle — Module de sérialisation d'objets Python	397
12.2	copyreg — Enregistre les fonctions support de pickle	410
12.3	shelve — Objet Python persistant	411
12.4	marshal — Internal Python object serialization	413
12.5	dbm — Interfaces to Unix « databases »	415
12.6	sqlite3 — Interface DB-API 2.0 pour bases de données SQLite	419
13	Compression de donnée et archivage	439

13.1	<code>zlib</code> — Compression compatible avec <code>gzip</code>	439
13.2	<code>gzip</code> — Support pour les fichiers <code>gzip</code>	443
13.3	<code>bz2</code> — Prise en charge de la compression <code>bzip2</code>	445
13.4	<code>lzma</code> — Compression via l'algorithme LZMA	448
13.5	<code>zipfile</code> — Travailler avec des archives ZIP	453
13.6	<code>tarfile</code> — Read and write tar archive files	461
14	Formats de fichiers	471
14.1	<code>csv</code> — Lecture et écriture de fichiers CSV	471
14.2	<code>configparser</code> — Configuration file parser	478
14.3	<code>netrc</code> — traitement de fichier <code>netrc</code>	495
14.4	<code>xdrllib</code> — Encode and decode XDR data	496
14.5	<code>plistlib</code> — Generate and parse Mac OS X <code>.plist</code> files	499
15	Service de cryptographie	503
15.1	<code>hashlib</code> — Algorithmes de hachage sécurisés et synthèse de messages	503
15.2	<code>hmac</code> — Authentification de messages par hachage en combinaison avec une clé secrète	514
15.3	<code>secrets</code> — Générer des nombres aléatoires de façon sécurisée pour la gestion des secrets	515
16	Services génériques du système d'exploitation	519
16.1	<code>os</code> — Diverses interfaces pour le système d'exploitation	519
16.2	<code>io</code> — Core tools for working with streams	562
16.3	<code>time</code> — Accès au temps et conversions	573
16.4	<code>argparse</code> — Parseur d'arguments, d'options, et de sous-commandes de ligne de commande	581
16.5	<code>getopt</code> — Analyseur de style C pour les options de ligne de commande	612
16.6	<code>logging</code> — Fonctionnalités de journalisation pour Python	614
16.7	<code>logging.config</code> — Logging configuration	629
16.8	<code>logging.handlers</code> — Logging handlers	638
16.9	Saisie de mot de passe portable	650
16.10	<code>curses</code> — Terminal handling for character-cell displays	650
16.11	<code>curses.textpad</code> — Text input widget for curses programs	667
16.12	<code>curses.ascii</code> — Utilities for ASCII characters	669
16.13	<code>curses.panel</code> — A panel stack extension for curses	671
16.14	<code>platform</code> — Access to underlying platform's identifying data	672
16.15	<code>errno</code> — Symboles du système <code>errno</code> standard	675
16.16	<code>ctypes</code> — Bibliothèque Python d'appels à des fonctions externes	681
17	Exécution concurrente	715
17.1	<code>threading</code> — Parallélisme basé sur les fils d'exécution (<i>threads</i>)	715
17.2	<code>multiprocessing</code> — Parallélisme par processus	727
17.3	Le paquet <code>concurrent</code>	768
17.4	<code>concurrent.futures</code> — Launching parallel tasks	768
17.5	<code>subprocess</code> — Gestion de sous-processus	774
17.6	<code>sched</code> — Event scheduler	789
17.7	<code>queue</code> — File synchronisée	791
17.8	<code>dummy_threading</code> — Module de substitution au module <code>threading</code>	793
17.9	<code>_thread</code> — API bas niveau de gestion de fils d'exécution	794
17.10	<code>_dummy_thread</code> — Module de substitution pour le module <code>_thread</code>	796
18	Communication et réseau entre processus	797
18.1	<code>socket</code> — Low-level networking interface	797
18.2	<code>ssl</code> — Emballage TLS/SSL pour les objets connecteurs	817
18.3	<code>select</code> — Waiting for I/O completion	847
18.4	<code>selectors</code> — High-level I/O multiplexing	854
18.5	<code>asyncio</code> — Entrées/Sorties asynchrone, boucle d'évènements, coroutines et tâches	857

18.6	<code>asyncore</code> — Gestionnaire de socket asynchrone	915
18.7	<code>asynchat</code> — Gestionnaire d'interfaces de connexion (<i>socket</i>) commande/réponse asynchrones	919
18.8	<code>signal</code> — Set handlers for asynchronous events	921
18.9	<code>mmap</code> — Memory-mapped file support	926
19	Traitement des données provenant d'Internet	931
19.1	<code>email</code> — Un paquet de gestion des e-mails et MIME	931
19.2	<code>json</code> — Encodage et décodage JSON	985
19.3	<code>mailcap</code> — Manipulation de fichiers Mailcap	994
19.4	<code>mailbox</code> — Manipuler les boîtes de courriels dans différents formats	995
19.5	<code>mimetypes</code> — Map filenames to MIME types	1012
19.6	<code>base64</code> — Encodages base16, base32, base64 et base85	1015
19.7	<code>binhex</code> — Encode et décode les fichiers <i>binhex4</i>	1018
19.8	<code>binascii</code> — Conversion entre binaire et ASCII	1019
19.9	<code>quopri</code> — Encode et décode des données <i>MIME quoted-printable</i>	1021
19.10	<code>uu</code> — Encode et décode les fichiers <i>uuencode</i>	1021
20	Outils de traitement de balises structurées	1023
20.1	<code>html</code> — Support du HyperText Markup Language	1023
20.2	<code>html.parser</code> — Simple HTML and XHTML parser	1024
20.3	<code>html.entities</code> — Définitions des entités HTML générales	1028
20.4	Modules de traitement XML	1029
20.5	<code>xml.etree.ElementTree</code> — The ElementTree XML API	1030
20.6	<code>xml.dom</code> — L'API Document Object Model	1045
20.7	<code>xml.dom.minidom</code> — Minimal DOM implementation	1056
20.8	<code>xml.dom.pulldom</code> — Support for building partial DOM trees	1060
20.9	<code>xml.sax</code> — Prise en charge des analyseurs SAX2	1062
20.10	<code>xml.sax.handler</code> — Base classes for SAX handlers	1064
20.11	<code>xml.sax.saxutils</code> — Utilitaires SAX	1069
20.12	<code>xml.sax.xmlreader</code> — Interface for XML parsers	1070
20.13	<code>xml.parsers.expat</code> — Fast XML parsing using Expat	1074
21	Gestion des protocoles internet	1085
21.1	<code>webbrowser</code> — Convenient Web-browser controller	1085
21.2	<code>cgi</code> — Common Gateway Interface support	1088
21.3	<code>cgitb</code> — Gestionnaire d'exceptions pour les scripts CGI	1095
21.4	<code>wsgiref</code> — Outils et implémentation de référence de WSGI	1095
21.5	<code>urllib</code> — Modules de gestion des URLs	1104
21.6	<code>urllib.request</code> — Extensible library for opening URLs	1104
21.7	<code>urllib.response</code> — Response classes used by urllib	1121
21.8	<code>urllib.parse</code> — Parse URLs into components	1122
21.9	<code>urllib.error</code> — Classes d'exceptions levées par <i>urllib.request</i>	1128
21.10	<code>urllib.robotparser</code> — Analyseur de fichiers <i>robots.txt</i>	1129
21.11	<code>http</code> — modules HTTP	1130
21.12	<code>http.client</code> — HTTP protocol client	1132
21.13	<code>ftplib</code> — FTP protocol client	1138
21.14	<code>poplib</code> — POP3 protocol client	1143
21.15	<code>imaplib</code> — IMAP4 protocol client	1146
21.16	<code>nntplib</code> — NNTP protocol client	1152
21.17	<code>smtplib</code> — SMTP protocol client	1159
21.18	<code>smtpd</code> — SMTP Server	1165
21.19	<code>telnetlib</code> — Telnet client	1168
21.20	<code>uuid</code> — UUID objects according to RFC 4122	1170
21.21	<code>socketserver</code> — A framework for network servers	1173

21.22	<code>http.server</code> — HTTP servers	1181
21.23	<code>http.cookies</code> — gestion d'état pour HTTP	1186
21.24	<code>http.cookiejar</code> — Cookie handling for HTTP clients	1190
21.25	<code>xmlrpc</code> — Modules Serveur et Client XMLRPC	1198
21.26	<code>xmlrpc.client</code> — XML-RPC client access	1198
21.27	<code>xmlrpc.server</code> — Basic XML-RPC servers	1205
21.28	<code>ipaddress</code> — IPv4/IPv6 manipulation library	1211
22	Services multimédia	1223
22.1	<code>audioloop</code> — Manipulation de données audio brutes	1223
22.2	<code>aifc</code> — Lis et écrit dans les fichiers AIFF et AIFC	1226
22.3	<code>sunau</code> — Read and write Sun AU files	1229
22.4	<code>wave</code> — Lecture et écriture des fichiers WAV	1231
22.5	<code>chunk</code> — Read IFF chunked data	1234
22.6	<code>colorsys</code> — Conversions entre les systèmes de couleurs	1235
22.7	<code>imghdr</code> — Determine the type of an image	1236
22.8	<code>sndhdr</code> — Détermine le type d'un fichier audio	1237
22.9	<code>ossaudiodev</code> — Access to OSS-compatible audio devices	1237
23	Internationalisation	1243
23.1	<code>gettext</code> — Services d'internationalisation multilingue	1243
23.2	<code>locale</code> — Services d'internationalisation	1251
24	Frameworks d'applications	1261
24.1	<code>turtle</code> — Tortue graphique	1261
24.2	<code>cmd</code> — Interpréteurs en ligne de commande.	1293
24.3	<code>shlex</code> — Simple lexical analysis	1298
25	Interfaces Utilisateur Graphiques avec Tk	1303
25.1	<code>tkinter</code> — Interface Python pour Tcl/Tk	1303
25.2	<code>tkinter.ttk</code> — Tk themed widgets	1314
25.3	<code>tkinter.tix</code> — Extension widgets for Tk	1330
25.4	<code>tkinter.scrolledtext</code> — Gadget texte avec barre de défilement	1335
25.5	<i>IDLE</i>	1336
25.6	Autres paquets d'interface graphique utilisateur	1346
26	Outils de développement	1347
26.1	<code>typing</code> — Support for type hints	1347
26.2	<code>pydoc</code> — Générateur de documentation et système d'aide en ligne	1362
26.3	<code>doctest</code> — Test interactive Python examples	1363
26.4	<code>unittest</code> — <i>Framework</i> de tests unitaires	1386
26.5	<code>unittest.mock</code> — Bibliothèque d'objets simulacres	1413
26.6	<code>unittest.mock</code> — getting started	1448
26.7	<code>2to3</code> — Traduction automatique de code en Python 2 vers Python 3	1468
26.8	<code>test</code> — Regression tests package for Python	1473
26.9	<code>test.support</code> — Utilities for the Python test suite	1476
27	Débogueur et instrumentation	1483
27.1	<code>bdb</code> — Framework de débogage	1483
27.2	<code>faulthandler</code> — Dump the Python traceback	1487
27.3	<code>pdb</code> — Le débogueur Python	1489
27.4	The Python Profilers	1495
27.5	<code>timeit</code> — Mesurer le temps d'exécution de fragments de code	1503
27.6	<code>trace</code> — Trace or track Python statement execution	1508
27.7	<code>tracemalloc</code> — Trace memory allocations	1510

28	Paquets et distribution de paquets logiciels	1521
28.1	distutils — Création et installation des modules Python	1521
28.2	ensurepip — Bootstrapping the pip installer	1522
28.3	venv — Création d’environnements virtuels	1523
28.4	zipapp — Manage executable python zip archives	1531
29	Environnement d’exécution Python	1539
29.1	sys — Paramètres et fonctions propres à des systèmes	1539
29.2	sysconfig — Provide access to Python’s configuration information	1554
29.3	builtins — Objets natifs	1557
29.4	__main__ — Point d’entrée des scripts	1558
29.5	warnings — Contrôle des alertes	1558
29.6	contextlib — Utilities for with-statement contexts	1563
29.7	abc — Classes de Base Abstraites	1575
29.8	atexit — Gestionnaire de fin de programme	1579
29.9	traceback — Print or retrieve a stack traceback	1581
29.10	__future__ — Définitions des futurs	1587
29.11	gc — Garbage Collector interface	1588
29.12	inspect — Inspect live objects	1591
29.13	site — Site-specific configuration hook	1605
29.14	fpectl — Floating point exception control	1608
30	Interpréteurs Python personnalisés	1611
30.1	code — Interpreter base classes	1611
30.2	codeop — Compilation de code Python	1613
31	Importer des modules	1615
31.1	zipimport — Importer des modules à partir d’archives Zip	1615
31.2	pkgutil — Package extension utility	1617
31.3	modulefinder — Identifie les modules utilisés par un script	1619
31.4	runpy — Locating and executing Python modules	1621
31.5	importlib — The implementation of import	1623
32	Services du Langage Python	1639
32.1	parser — Accès aux arbres syntaxiques	1639
32.2	ast — Arbres Syntaxiques Abstraits	1643
32.3	symtable — Access to the compiler’s symbol tables	1649
32.4	symbol — Constantes utilisées dans les Arbres Syntaxiques	1651
32.5	token — Constantes utilisées avec les arbres d’analyse Python (<i>parse trees</i>)	1651
32.6	keyword — Tester si des chaînes sont des mot-clés Python	1653
32.7	tokenize — Analyseur lexical de Python	1653
32.8	tabnanny — Détection d’indentation ambiguë	1657
32.9	pyclbr — Python class browser support	1657
32.10	py_compile — Compile Python source files	1659
32.11	compileall — Byte-compile Python libraries	1660
32.12	dis — Désassembleur pour le code intermédiaire de Python	1662
32.13	pickletools — Tools for pickle developers	1675
33	Services divers	1677
33.1	formatter — Generic output formatting	1677
34	Services spécifiques à MS Windows	1683
34.1	msilib — Read and write Microsoft Installer files	1683
34.2	msvcrt — Useful routines from the MS VC++ runtime	1689
34.3	winreg — Windows registry access	1690

34.4	<code>winsound</code> — Sound-playing interface for Windows	1698
35	Services spécifiques à Unix	1701
35.1	<code>posix</code> — Les appels système POSIX les plus courants	1701
35.2	<code>pwd</code> — The password database	1702
35.3	<code>spwd</code> — La base de données de mots de passe <i>shadow</i>	1703
35.4	<code>grp</code> — The group database	1704
35.5	<code>crypt</code> — Function to check Unix passwords	1705
35.6	<code>termios</code> — Le style POSIX le contrôle TTY	1706
35.7	<code>tty</code> — Fonctions de gestion du terminal	1708
35.8	<code>pty</code> — Outils de manipulation de pseudo-terminaux	1708
35.9	<code>fcntl</code> — The <code>fcntl</code> and <code>ioctl</code> system calls	1709
35.10	<code>pipes</code> — Interface au <i>pipelines</i> shell	1711
35.11	<code>resource</code> — Resource usage information	1712
35.12	<code>nis</code> — Interface à Sun's NIS (pages jaunes)	1716
35.13	<code>syslog</code> — Unix syslog library routines	1717
36	Modules remplacés	1719
36.1	<code>optparse</code> — Parser for command line options	1719
36.2	<code>imp</code> — Access the import internals	1745
37	Modules non Documentés	1751
37.1	Modules spécifiques à une plateforme	1751
A	Glossaire	1753
B	À propos de ces documents	1765
B.1	Contributeurs de la documentation Python	1765
C	Histoire et licence	1767
C.1	Histoire du logiciel	1767
C.2	Conditions générales pour accéder à, ou utiliser, Python	1768
C.3	Licences et remerciements pour les logiciels tiers	1771
D	Copyright	1785
	Bibliographie	1787
	Index des modules Python	1789
	Index	1793

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

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

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

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

CHAPITRE 1

Introduction

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

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

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

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

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

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

Que le spectacle commence !

CHAPITRE 2

Fonctions natives

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

Fonctions natives				
<i>abs()</i>	<i>dict()</i>	<i>help()</i>	<i>min()</i>	<i>setattr()</i>
<i>all()</i>	<i>dir()</i>	<i>hex()</i>	<i>next()</i>	<i>slice()</i>
<i>any()</i>	<i>divmod()</i>	<i>id()</i>	<i>object()</i>	<i>sorted()</i>
<i>ascii()</i>	<i>enumerate()</i>	<i>input()</i>	<i>oct()</i>	<i>staticmethod()</i>
<i>bin()</i>	<i>eval()</i>	<i>int()</i>	<i>open()</i>	<i>str()</i>
<i>bool()</i>	<i>exec()</i>	<i>isinstance()</i>	<i>ord()</i>	<i>sum()</i>
<i>bytearray()</i>	<i>filter()</i>	<i>issubclass()</i>	<i>pow()</i>	<i>super()</i>
<i>bytes()</i>	<i>float()</i>	<i>iter()</i>	<i>print()</i>	<i>tuple()</i>
<i>callable()</i>	<i>format()</i>	<i>len()</i>	<i>property()</i>	<i>type()</i>
<i>chr()</i>	<i>frozenset()</i>	<i>list()</i>	<i>range()</i>	<i>vars()</i>
<i>classmethod()</i>	<i>getattr()</i>	<i>locals()</i>	<i>repr()</i>	<i>zip()</i>
<i>compile()</i>	<i>globals()</i>	<i>map()</i>	<i>reversed()</i>	<i>__import__()</i>
<i>complex()</i>	<i>hasattr()</i>	<i>max()</i>	<i>round()</i>	
<i>delattr()</i>	<i>hash()</i>	<i>memoryview()</i>	<i>set()</i>	

abs(*x*)

Donne la valeur absolue d'un nombre. L'argument peut être un nombre entier ou un nombre à virgule flottante. Si l'argument est un nombre complexe, son `module` est donné.

all(*iterable*)

Donne True si tous les éléments de *iterable* sont vrais (ou s'il est vide), équivaut à :

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

any (*iterable*)

Donne True si au moins un élément de *iterable* est vrai. Faux est aussi donné dans le cas où *iterable* est vide, équivaut à :

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

ascii (*object*)

Donne, tout comme `repr()`, une chaîne contenant une représentation affichable d'un objet, en transformant les caractères non ASCII donnés par `repr()` en utilisant des séquences d'échappement `\x`, `\u` ou `\U`. Cela génère une chaîne similaire à ce que renvoie `repr()` dans Python 2.

bin (*x*)

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

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

Que le préfixe `0b` soit souhaité ou non, vous pouvez utiliser les moyens suivants.

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

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

class bool (*[x]*)

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

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

Donne un nouveau tableau d'octets. La classe `bytearray` est une séquence muable de nombre entiers dans l'intervalle $0 \leq x < 256$. Il possède la plupart des méthodes des séquences variables, décrites dans *Types de séquences muables*, ainsi que la plupart des méthodes de la classe `bytes`, voir *Opérations sur les bytes et bytearray*.

Le paramètre optionnel *source* peut être utilisé pour initialiser l'array de quelques manières différentes :

- Si c'est une *chaîne*, vous devez aussi donner les paramètres *encoding* pour l'encodage (et éventuellement *errors*). La fonction `bytearray()` convertit ensuite la chaîne en `bytes` via la méthode `str.encode()`.
- Si c'est un *entier*, l'array aura cette taille et sera initialisé de *null bytes*.
- Si c'est un objet conforme à l'interface *buffer*, un *buffer* en lecture seule de l'objet sera utilisé pour initialiser l'array.
- Si c'est un *itérable*, il doit itérer sur des nombres entiers dans l'intervalle $0 \leq x < 256$, qui seront utilisés pour initialiser le contenu de l'array.

Sans argument, un array de taille vide est créé.

Voir *Séquences Binaires — bytes, bytearray, memoryview* et *Objets bytearray*.

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

Donne un nouvel objet `bytes`, qui est une séquence immuable de nombre entiers dans l'intervalle $0 \leq x \leq 256$. Les `bytes` est une version immuable de `bytearray` – avec les mêmes méthodes d'accès, et le même comportement lors de l'indexation ou la découpe.

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

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

Voir aussi *Séquences Binaires* — `bytes`, `bytearray`, `memoryview`, *Objets bytes*, et *Opérations sur les bytes et bytearray*.

callable (*object*)

Donne `True` si l'argument *object* semble être callable, sinon `False`. Lorsqu'elle donne vrai, il reste une éventualité pour qu'un appel échoue, cependant, lorsqu'elle donne faux, il ne sera jamais possible d'appeler *object*. Notez que les classes sont callable (appeler une classe donne une nouvelle instance). Les instances sont callable si leur classe définit une méthode `__call__()`.

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

chr (*i*)

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

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

@classmethod

Transforme une méthode en méthode de classe.

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

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

La forme `@classmethod` est un *decorator* – consultez la documentation sur la définition de fonctions dans fonction pour plus de détails.

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

Les méthodes de classe sont différentes des méthodes statiques du C++ ou du Java. Si c'est elles dont vous avez besoin, regardez du côté de `staticmethod()`.

Pour plus d'informations sur les méthodes de classe, consultez la documentation sur la hiérarchie des types standards dans types.

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

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

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

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

The optional arguments *flags* and *dont_inherit* control which future statements affect the compilation of *source*. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling `compile()`. If the *flags* argument is given and *dont_inherit* is not (or is zero) then the future statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it – the future statements in effect around the call to `compile` are ignored.

Les instructions futures sont spécifiées par des bits, il est ainsi possible d'en spécifier plusieurs en les combinant avec un *ou* binaire. Les bits requis pour spécifier une certaine fonctionnalité se trouvent dans l'attribut `compiler_flag` de la classe `Feature` du module `__future__`.

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

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

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

Note : Lors de la compilation d'une chaîne de plusieurs lignes de code avec les modes 'single' ou 'eval', celle-ci doit être terminée d'au moins un retour à la ligne. Cela permet de faciliter la distinction entre les instructions complètes et incomplètes dans le module *code*.

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

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

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

class complex ([*real* [, *imag*]])

Donne un nombre complexe de valeur *real* + *imag**1j, ou convertit une chaîne ou un nombre en nombre complexe. Si le premier paramètre est une chaîne, il sera interprété comme un nombre complexe et la fonction doit être appelée dans second paramètre. Le second paramètre ne peut jamais être une chaîne. Chaque argument peut être de n'importe quel type numérique (même complexe). Si *imag* est omis, sa valeur par défaut est zéro, le constructeur effectue alors une simple conversion numérique comme le font *int* ou *float*. Si aucun argument n'est fourni, donne 0j.

Note : Lors de la conversion depuis une chaîne, elle ne doit pas contenir d'espaces autour des opérateurs binaires + ou -. Par exemple `complex('1+2j')` est bon, mais `complex('1 + 2j')` lève une *ValueError*.

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

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

delattr (*object*, *name*)

C'est un cousin de `setattr()`. Les arguments sont un objet et une chaîne. La chaîne doit être le nom de l'un des attributs de l'objet. La fonction supprime l'attribut nommé, si l'objet l'y autorise. Par exemple `delattr(x, 'foobar')` est l'équivalent de `del x.foobar`.

class dict (***kwarg*)

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

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

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

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

dir ([*object*])

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

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

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

Le mécanisme par défaut de `dir()` se comporte différemment avec différents types d'objets, car elle préfère donner une information pertinente plutôt qu'exhaustive :

- Si l'objet est un module, la liste contiendra les noms des attributs du module.
- Si l'objet est un type ou une classe, la liste contiendra les noms de ses attributs, et récursivement, des attributs de ses parents.
- Autrement, la liste contient les noms des attributs de l'objet, le nom des attributs de la classe, et récursivement des attributs des parents de la classe.

La liste donnée est triée par ordre alphabétique, par exemple :

```
>>> import struct
>>> dir()      # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct) # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '__clearcache', 'calcsizes', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

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

divmod(*a*, *b*)

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

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

Donne un objet énumérant. *iterable* doit être une séquence, un *iterator*, ou tout autre objet supportant l'itération. La méthode `__next__()` de l'itérateur donné par `enumerate()` donne un tuple contenant un compte (démarrant à *start*, 0 par défaut) et les valeurs obtenues de l'itération sur *iterable*.

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

Équivalent à :

```
def enumerate(sequence, start=0):
    n = start
```

(suite sur la page suivante)

(suite de la page précédente)

```

for elem in sequence:
    yield n, elem
    n += 1

```

eval (*expression*, *globals*=None, *locals*=None)

Les arguments sont : une chaîne, et optionnellement des locales et des globales. S'il est fourni, *globals* doit être un dictionnaire. S'il est fourni, *locals* peut être n'importe quel objet *mapping*.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module *builtins* is inserted under that key before *expression* is parsed. This means that *expression* normally has full access to the standard *builtins* module and restricted environments are propagated. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where *eval()* is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example :

```

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

```

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

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

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

exec (*object*[, *globals*[, *locals*]])

Cette fonction permet l'exécution dynamique de code Python. *object* doit être soit une chaîne soit un objet code. Si c'est une chaîne, elle est d'abord analysée en une suite d'instructions Python qui sont ensuite exécutés (sauf erreur de syntaxe).¹ Si c'est un objet code, il est simplement exécuté. Dans tous les cas, le code fourni doit être valide selon les mêmes critères que s'il était un script dans un fichier (voir la section « File Input » dans le manuel). Gardez en tête que les mots clefs *return* et *yield* ne peuvent pas être utilisés en dehors d'une fonction, même dans du code passé à *exec()*. La fonction donne None.

Dans tous les cas, si les arguments optionnels sont omis, le code est exécuté dans le contexte actuel. Si seul *globals* est fourni, il doit être un dictionnaire qui sera utilisé pour les globales et les locales. Si les deux sont fournis, ils sont utilisés respectivement pour les variables globales et locales. *locales* peut être n'importe quel objet *mapping*. Souvenez vous qu'au niveau d'un module, les dictionnaires des locales et des globales ne sont qu'un. Si *exec* reçoit deux objets distincts dans *globals* et *locals*, le code sera exécuté comme s'il était inclus dans une définition de classe. Si le dictionnaire *globals* ne contient pas de valeur pour la clef `__builtins__`, une référence au dictionnaire du module *builtins* y est inséré. Cela vous permet de contrôler quelles fonctions natives sont exposées au code exécuté en insérant votre propre dictionnaire `__builtins__` dans *globals* avant de le donner à *exec()*.

Note : Les fonctions natives *globals()* et *locals()* donnent respectivement les dictionnaires globaux et locaux, qui peuvent être utiles en deuxième et troisième argument de *exec()*.

Note : La valeur par défaut pour *locals* se comporte comme la fonction *locals()* : Il est déconseillé de modifier le dictionnaire *locals* par défaut. Donnez un dictionnaire explicitement à *locals* si vous désirez observer l'effet du

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

code sur les variables locales, après que `exec()` soit terminée.

filter (*function*, *iterable*)

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

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

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

class float ([*x*])

Donne un nombre à virgule flottante depuis un nombre ou une chaîne *x*.

Si l'argument est une chaîne, elle devrait contenir un nombre décimal, éventuellement précédé d'un signe, et pouvant être entouré d'espaces. Le signe optionnel peut être '+' ou '-'. Un signe '+' n'a pas d'effet sur la valeur produite. L'argument peut aussi être une chaîne représentant un NaN (*Not-a-Number*), l'infini positif, ou l'infini négatif. Plus précisément, l'argument doit se conformer à la grammaire suivante, après que les espaces en début et fin de chaîne aient été retirés :

```
sign          ::= "+" | "-"
infinity      ::= "Infinity" | "inf"
nan           ::= "nan"
numeric_value ::= floatnumber | infinity | nan
numeric_string ::= [sign] numeric_value
```

Ici `floatnumber` est un nombre à virgule flottante littéral Python, décrit dans `floating`. La casse n'y est pas significative, donc, par exemple, `"inf"`, `" Inf"`, `"INFINITY"`, et `" iNfInItY"` sont tous des orthographes valides pour un infini positif.

Autrement, si l'argument est un entier ou un nombre à virgule flottante, un nombre à virgule flottante de même valeur (en accord avec la précision des nombres à virgule flottante de Python) est donné. Si l'argument est en dehors de l'intervalle d'un nombre à virgule flottante pour Python, `OverflowError` est levée.

Pour un objet Python *x*, `float(x)` est délégué à `x.__float__()`.

Dans argument, `0.0` est donné.

Exemples :

```
>>> float('+1.23')
1.23
>>> float(' -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

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

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

format (*value*[, *format_spec*])

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

Par défaut, *format_spec* est une chaîne vide qui généralement donne le même effet qu'appeler *str(value)*.

Un appel à *format(value, format_spec)* est transformé en *type(value).__format__(value, format_spec)*, qui contourne le dictionnaire de l'instance lors de la recherche de la méthode *__format__()*. Une exception *TypeError* est levée si la recherche de la méthode atteint *object* et que *format_spec* n'est pas vide, ou si soit *format_spec* soit la le résultat ne sont pas des chaînes.

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

class frozenset ([*iterable*])

Donne un nouveau *frozenset*, dont les objets sont éventuellement tirés d'*iterable*. *frozenset* est une classe native. Voir *frozenset* et *Types d'ensembles — set, frozenset* pour leurs documentation.

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

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

Donne la valeur de l'attribut nommé *name* de l'objet *object*. *name* doit être une chaîne. Si la chaîne est le nom d'un des attributs de l'objet, le résultat est la valeur de cet attribut. Par exemple, *getattr(x, 'foobar')* est équivalent à *x.foobar*. Si l'attribut n'existe pas, et que *default* est fourni, il est renvoyé, sinon l'exception *AttributeError* est levée.

globals ()

Donne une représentation de la table de symboles globaux sous forme d'un dictionnaire. C'est toujours le dictionnaire du module courant (dans une fonction ou méthode, c'est le module où elle est définie, et non le module d'où elle est appelée).

hasattr (*object*, *name*)

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

hash (*object*)

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

Note : Pour les objets dont la méthode *__hash__()* est implémentée, notez que *hash()* tronque la valeur donnée en fonction du nombre de bits de la machine hôte. Voir *__hash__()* pour plus d'informations.

help ([*object*])

Invoque le système d'aide natif. (Cette fonction est destinée à l'usage en mode interactif.) Soit aucun argument n'est fourni, le système d'aide démarre dans l'interpréteur. Si l'argument est une chaîne, un module, une fonction, une classe, une méthode, un mot clef, ou un sujet de documentation pourtant ce nom est recherché, et une page d'aide est affichée sur la console. Si l'argument est d'un autre type, une page d'aide sur cet objet est générée.

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

Modifié dans la version 3.4 : Les changements aux modules *pydoc* et *inspect* rendent les signatures des appelables plus compréhensible et cohérente.

hex (*x*)

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

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

Si vous voulez convertir un nombre entier en chaîne hexadécimale, en majuscule ou non, préfixée ou non, vous pouvez utiliser les moyens suivants :

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

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

Voir aussi `int()` pour convertir une chaîne hexadécimale en un entier en lui spécifiant 16 comme base.

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

`id(object)`

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

CPython implementation detail : This is the address of the object in memory.

`input([prompt])`

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

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

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

`class int(x=0)`

`class int(x, base=10)`

Donne un entier construit depuis un nombre ou une chaîne *x*, ou 0 si aucun argument n'est fourni. Si *x* définit une méthode `__int__()`, `int(x)` renvoie `x.__int__()`. Si *x* définit `__trunc__()`, `int(x)` renvoie `x.__trunc__()`. Les nombres à virgule flottante sont tronqués vers zéro.

Si *x* n'est pas un nombre ou si *base* est fourni, alors *x* doit être une chaîne, un `bytes`, ou un `bytearray` représentant un entier littéral de base *base*. Le littéral peut être précédé d'un + ou d'un - (sans être séparés par un espace), et peut être entouré d'espaces. Un littéral de base *n* est composé des symboles de 0 à *n*-1 où *a* jusqu'à *z* (ou A à Z) représentent les valeurs de 10 à 35. La *base* par défaut est 10. Les valeurs autorisées pour *base* sont 0 et 2–36. Les littéraux en base 2, 8, et 16 peuvent être préfixés avec `0b/0B`, `0o/0O`, ou `0x/0X` tout comme les littéraux dans le code. Fournir 0 comme *base* demande d'interpréter exactement comme un littéral dans Python, donc la base sera 2, 8, 10, ou 16, ainsi `int('010', 0)` n'est pas légal, alors que `int('010')` l'est tout comme `int('010', 8)`.

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

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

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

`isinstance(object, classinfo)`

Donne True si *object* est une instance de *classinfo*, ou d'une de ses classes filles, directe, indirecte, ou *virtuelle*. Si

object n'est pas un objet du type donné, la fonction donne toujours `False`. Si *classinfo* est un *tuple* de types (ou récursivement, d'autres *tuples*), donne `True` si *object* est une instance de n'importe quel de ces types. Si *classinfo* n'est ni un type ni un *tuple* de types (et récursivement), une exception `TypeError` est levée.

issubclass (*class*, *classinfo*)

Donne `True` si *class* est une classe fille (directe, indirecte, ou *virtual*) de *classinfo*. Une classe est considérée sous-classe d'elle même. *classinfo* peut être un *tuple* de classes, dans ce cas la vérification sera faite pour chaque classe de *classinfo*. Dans tous les autres cas, `TypeError` est levée.

iter (*object* [, *sentinel*])

Donne un objet *iterator*. Le premier argument est interprété très différemment en fonction de la présence du second argument. Sans second argument, *object* doit être une collection d'objets supportant le protocole d'itération (la méthode `__iter__()`), ou supportant le protocole des séquences (la méthode `getitem()`, avec des nombres entiers commençant par 0 comme argument). S'il ne supporte aucun de ces protocoles, `TypeError` est levée. Si le second argument *sentinel* est fourni, *objet* doit être callable. L'itérateur créé dans ce cas appellera *object* dans argument à chaque appel de `__next__()`, si la valeur reçue est égale à *sentinel* `StopIteration` est levée, autrement la valeur est donnée.

Voir aussi *Les types itérateurs*.

Une autre application utile de la deuxième forme de `iter()` est de lire les lignes d'un fichier jusqu'à ce qu'une certaine ligne soit atteinte. L'exemple suivant lis un fichier jusqu'à ce que `readline()` donne une ligne vide :

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

len (*s*)

Donne la longueur (nombre d'éléments) d'un objet. L'argument peut être une séquence (tel qu'une chaîne, un objet bytes, tuple, list ou range) ou une collection (tel qu'un dict, set ou frozenset).

class list ([*iterable*])

Plutôt qu'être une fonction, *list* est en fait un type de séquence variable, tel que documenté dans *Listes* et *Types séquentiels* — *list*, *tuple*, *range*.

locals ()

Met à jour et donne un dictionnaire représentant la table des symboles locaux. Les variables libres sont données par `locals()` lorsqu'elle est appelée dans le corps d'une fonction, mais pas dans le corps d'une classe.

Note : Le contenu de ce dictionnaire ne devrait pas être modifié, les changements peuvent ne pas affecter les valeurs des variables locales ou libres utilisées par l'interpréteur.

map (*function*, *iterable*, ...)

Donne un itérateur appliquant *function* à chaque élément de *iterable*, et donnant ses résultats au fur et à mesure avec `yield`. Si d'autres *iterable* sont fournis, *function* doit prendre autant d'arguments, et sera appelée avec les éléments de tous les itérables en parallèle. Avec plusieurs itérables, l'itération s'arrête avec l'itérable le plus court. Pour les cas où les arguments seraient déjà rangés sous forme de tuples, voir `itertools.starmap()`.

max (*iterable*, *[, *key*, *default*])

max (*arg1*, *arg2*, **args* [, *key*])

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

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

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

Si plusieurs éléments représentent la plus grande valeur, le premier rencontré est donné. C'est cohérent avec d'autres outils préservant une stabilité lors du tri, tel que `sorted(iterable, key=keyfunc,`

`reverse=True)[0]` et `heapq.nlargest(1, iterable, key=keyfunc)`.

Nouveau dans la version 3.4 : L'argument exclusivement par mot clef *default*.

memoryview (*obj*)

Donne une « vue mémoire » (*memory view*) créée depuis l'argument. Voir [Vues de mémoires](#) pour plus d'informations.

min (*iterable*, *[, *key*, *default*])

min (*arg1*, *arg2*, **args*[, *key*])

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

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

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

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

Nouveau dans la version 3.4 : L'argument exclusivement par mot clef *default*.

next (*iterator*[, *default*])

Donne l'élément suivant d'*iterator* en appelant sa méthode `__next__()`. Si *default* est fourni, il sera donné si l'itérateur est épuisé, sinon *StopIteration* est levée.

class object

Donne un objet vide. *object* est la classe parente de toute les classes. C'est elle qui porte les méthodes communes à toutes les instances de classes en Python. Cette fonction n'accepte aucun argument.

Note : *object* n'a pas d'attribut `__dict__`, vous ne pouvez donc pas assigner d'attributs arbitraire à une instance d'*object*.

oct (*x*)

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

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

Si vous voulez convertir un nombre entier en chaîne octale, avec ou sans le préfixe `0o`, vous pouvez utiliser les moyens suivants.

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

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

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

Ouvre *file* et donne un *file object* correspondant. Si le fichier ne peut pas être ouvert, une *OSError* est levée.

file est un *path-like object* donnant le chemin (absolu ou relatif au répertoire courant) du fichier à ouvrir ou un nombre entier représentant le descripteur de fichier à envelopper. (Si un descripteur de fichier est donné, il sera fermé en même temps que l'objet *I/O* renvoyé, sauf si *closefd* est mis à *False*.)

mode est une chaîne optionnelle permettant de spécifier dans quel mode le fichier est ouvert. Par défaut, *mode* vaut *'r'* qui signifie « ouvrir en lecture pour du texte ». *'w'* est aussi une valeur classique, permettant d'écrire (vidant le fichier s'il existe), ainsi que *'x'* permettant une création exclusive et *'a'* pour ajouter à la fin du fichier (qui sur certains systèmes Unix signifie que toutes les écritures seront des ajouts en fin de fichier peu importe la position demandée). En mode texte, si *encoding* n'est pas spécifié l'encodage utilisé est dépendant de la plateforme : `locale.getpreferredencoding(False)` est appelée pour obtenir l'encodage de la locale actuelle. (Pour lire et écrire des octets bruts, utilisez le mode binaire en laissant *encoding* non spécifié.) Les modes disponibles sont :

Caractère	Signification
<i>'r'</i>	ouvre en lecture (par défaut)
<i>'w'</i>	ouvre en écriture, tronquant le fichier
<i>'x'</i>	ouvre pour une création exclusive, échouant si le fichier existe déjà
<i>'a'</i>	ouvre en écriture, ajoutant à la fin du fichier s'il existe
<i>'b'</i>	mode binaire
<i>'t'</i>	mode texte (par défaut)
<i>'+'</i>	ouvre un fichier pour le modifier (lire et écrire)
<i>'U'</i>	mode <i>universal newlines</i> (obsolète)

Les mode par défaut est *'r'* (ouvrir pour lire du texte, synonyme de *'rt'*). Pour un accès en lecture écriture binaire, le mode *'w+b'* ouvre et vide le fichier. *'r+b'* ouvre le fichier sans le vider.

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

Note : Python ne dépend pas de l'éventuelle notion de fichier texte du système sous-jacent, tout le traitement est effectué par Python lui même, et est ainsi indépendant de la plateforme.

buffering est un entier optionnel permettant de configurer l'espace tampon. Donnez 0 pour désactiver l'espace tampon (seulement autorisé en mode binaire), 1 pour avoir un *buffer* travaillant ligne par ligne (seulement disponible en mode texte), ou un entier supérieur à 1 pour donner la taille en octets d'un tampon de taille fixe. Sans l'argument *buffering*, les comportements par défaut sont les suivants :

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

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

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

- *'strict'* pour lever une *ValueError* si une erreur d'encodage est rencontrée. La valeur par défaut, *None*, a le même effet.
- *'ignore'* ignore les erreurs. Notez qu'ignorer les erreurs d'encodage peut mener à des pertes de données.
- *'replace'* insère un marqueur de substitution (tel que *'?'*) en place des données mal formées.
- *'surrogateescape'* représentera chaque octet incorrect par un code caractère de la zone *Private Use Area* d'Unicode, de *U+DC80* à *U+DCFF*. Ces codes caractères privés seront ensuite transformés dans les mêmes

octets erronés si le gestionnaire d'erreur `surrogateescape` est utilisé lors de l'écriture de la donnée. C'est utile pour traiter des fichiers d'un encodage inconnu.

- `'xmlcharrefreplace'` est seulement supporté à l'écriture vers un fichier. Les caractères non gérés par l'encodage sont remplacés par une référence de caractère XML `&#nnn;`.
- `'backslashreplace'` remplace les données mal formées par des séquences d'échappement Python (utilisant des *backslash*).
- `'namereplace'` (aussi supporté lors de l'écriture) remplace les caractères non supportés par des séquences d'échappement `\N{...}`.

`newline` contrôle comment le mode *universal newlines* fonctionne (seulement en mode texte). Il peut être `None`, `'\n'`, `'\r'`, et `'\r\n'`. Il fonctionne comme suit :

- Lors de la lecture, si `newline` est `None`, le mode *universal newlines* est activé. Les lignes lues peuvent terminer par `'\n'`, `'\r'`, ou `'\r\n'`, qui sont remplacés par `'\n'`, avant d'être données à l'appelant. S'il vaut `'*'`, le mode *universal newlines* est activé mais les fin de lignes ne sont pas remplacés. S'il a n'importe quel autre valeur autorisée, les lignes sont seulement terminées par la chaîne donnée, qui est rendue tel qu'elle.
- Lors de l'écriture, si `newline` est `None`, chaque `'\n'` est remplacé par le séparateur de lignes par défaut du système `os.linesep`. Si `newline` est `*` ou `'\n'` aucun remplacement n'est effectué. Si `newline` est un autre caractère valide, chaque `'\n'` sera remplacé par la chaîne donnée.

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

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

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

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

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

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

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

Modifié dans la version 3.3 :

- Le paramètre `opener` a été ajouté.
- Le mode `'x'` a été ajouté.
- `IOError` était normalement levée, elle est maintenant un alias de `OSError`.
- `FileExistsError` est maintenant levée si le fichier ouvert en mode création exclusive (`'x'`) existe déjà.

Modifié dans la version 3.4 :

- Il n'est plus possible d'hériter de `file`.

Deprecated since version 3.4, will be removed in version 4.0 : Le mode 'U'.

Modifié dans la version 3.5 :

- Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).
- Le gestionnaire d'erreurs 'namereplace' a été ajouté.

Modifié dans la version 3.6 :

- Ajout du support des objets implémentant `os.PathLike`.
- Sous Windows, ouvrir un *buffer* du terminal peut renvoyer une sous-classe de `io.RawIOBase` autre que `io.FileIO`.

ord(c)

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

pow(x, y[, z])

Donne x puissance y , et si z est présent, donne x puissance y modulo z (calculé de manière plus efficace que `pow(x, y) % z`). La forme à deux arguments est équivalent à `x**y`.

Les arguments doivent être de types numériques. Avec des opérandes de différents types, les mêmes règles de coercition que celles des opérateurs arithmétiques binaires s'appliquent. Pour des opérandes de type `int`, le résultat sera de même type que les opérandes (après coercition) sauf si le second argument est négatif, dans ce cas, les arguments sont convertis en `float`, et le résultat sera un `float` aussi. Par exemple, `10**2` donne 100, alors que `10**-2` donne 0.01. Si le second argument est négatif, le troisième doit être omis. Si z est fourni, x et y doivent être des entiers et y positif.

print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Écrit *objects* dans le flux texte *file*, séparés par *sep* et suivis de *end*. *sep*, *end*, *file*, et *flush*, s'ils sont présents, doivent être données par mot clef.

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

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

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

Modifié dans la version 3.3 : Ajout de l'argument par mot clef *flush*.

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

Donne un attribut propriété.

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

Une utilisation typique : définir un attribut managé x :

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

    def getx(self):
        return self._x

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

(suite sur la page suivante)

(suite de la page précédente)

```
def delx(self):
    del self._x

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

Si *c* est une instance de *C*, *c.x* appellera le *getter*, *c.x* = *value* invoquera le *setter*, et *del x* le *deleter*. S'il est donné, *doc* sera la *docstring* de l'attribut. Autrement la propriété copiera celle de *fget* (si elle existe). Cela rend possible la création de propriétés en lecture seule en utilisant simplement *property()* comme un *decorator* :

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

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

Le décorateur *@property* transforme la méthode *voltage()* en un *getter* d'un attribut du même nom, et donne « *Get the current voltage* » comme *docstring* de *voltage*.

Un objet propriété a les méthodes *getter*, *setter* et *deleter* utilisables comme décorateurs créant une copie de la propriété avec les accesseurs correspondants définis par la fonction de décoration. C'est plus clair avec un exemple :

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

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

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

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

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

L'objet propriété donné a aussi les attributs *fget*, *fset* et *fdel* correspondant correspondants aux arguments du constructeur.

Modifié dans la version 3.5 : Les *docstrings* des objets propriété peuvent maintenant être écrits.

range (*stop*)

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

Plutôt qu'être une fonction, *range* est en fait une séquence immuable, tel que documenté dans *Ranges* et *Types séquentiels* — *list*, *tuple*, *range*.

repr (*object*)

Donne une chaîne contenant une représentation affichable de l'objet. Pour de nombreux types, cette fonction essaye de donner une chaîne qui donnera à son tour un objet de même valeur lorsqu'elle est passée à *eval()*, sinon la représentation sera une chaîne entourée de chevrons contenant le nom du type et quelques informations supplé-

mentaires souvent le nom et l'adresse de l'objet. Une classe peut contrôler ce que cette fonction donne pour ses instances en définissant une méthode `__repr__()`.

reversed (*seq*)

Donne un *iterator* inversé. *seq* doit être un objet ayant une méthode `__reverse__()` ou supportant le protocole séquence (la méthode `__len__()` et la méthode `__getitem__()` avec des arguments entiers commençant à zéro).

round (*number* [, *ndigits*])

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

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

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

class set ([*iterable*])

Donne un nouveau *set*, dont les éléments peuvent être extraits d'*iterable*. *set* est une classe native. Voir *set* et *Types d'ensembles — set, frozenset* pour la documentation de cette classe.

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

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

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

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

Donne un objet *slice* représentant un ensemble d'indices spécifiés par `range(start, stop, step)`. Les arguments *start* et *step* valent `None` par défaut. Les objets *slice* (tranches) ont les attributs suivants en lecture seule : *start*, *stop*, et *step* qui valent simplement les trois arguments (ou leurs valeur par défaut). Ils n'ont pas d'autres fonctionnalité explicite, cependant ils sont utilisés par *Numerical Python* et d'autres bibliothèques tierces. Les objets *slice* sont aussi générés par la notation par indices étendue. Par exemple `a[start:stop:step]` ou `a[start:stop, i]`. Voir `itertools.islice()` pour une version alternative donnant un itérateur.

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

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

A deux arguments optionnels qui doivent être fournis par mot clef.

key specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

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

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

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

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

@staticmethod

Transforme une méthode en méthode statique.

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

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

La forme `@staticmethod` est un *decorator* de fonction. Voir la description des définitions de fonction dans fonction pour plus de détails.

Elle peut être appelée soit sur une classe (tel que `C.f()`) ou sur une instance (tel que `C().f()`). L'instance est ignorée, sauf pour sa classe.

Les méthodes statiques en Python sont similaires à celles trouvées en Java ou en C++. Consultez `classmethod()` pour une variante utile pour créer des constructeurs alternatifs.

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

```
class C:
    builtin_open = staticmethod(open)
```

Pour plus d'informations sur les méthodes statiques, consultez la documentation de la hiérarchie des types standards dans types.

class str (*object*="")

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

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

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

sum (*iterable*[, *start*])

Additionne *start* et les éléments d'*iterable* de gauche à droite et en donne le total. *start* vaut 0 par défaut. Les éléments d'*iterable* sont normalement des nombres, et la valeur de *start* ne peut pas être une chaîne.

Pour certains cas, il existe de bonnes alternatives à `sum()`. La bonne méthode, et rapide, de concaténer une séquence de chaînes est d'appeler `' '.join(séquence)`. Pour additionner des nombres à virgule flottante avec une meilleure précision, voir `math.fsum()`. Pour concaténer une série d'itérables, utilisez plutôt `itertools.chain()`.

super ([*type*[, *object-or-type*]])

Donne un objet mandataire (*proxy object* en anglais) déléguant les appels de méthode à une classe parente ou sœur de type *type*. C'est utile pour accéder à des méthodes héritées et substituées dans la classe. L'ordre de recherche est le même que celui utilisé par `getattr()` sauf que *type* lui-même est sauté.

L'attribut `__mro__` de *type* liste l'ordre de recherche de la méthode de résolution utilisée par `getattr()` et `super()`. L'attribut est dynamique et peut changer lorsque la hiérarchie d'héritage est modifiée.

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

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

Le second est la gestion d'héritage multiple coopératif dans un environnement d'exécution dynamique. Cet usage est unique à Python, il ne se retrouve ni dans les langages compilés statiquement, ni dans les langages ne gérant que l'héritage simple. Cela rend possible d'implémenter un héritage en diamant dans lequel plusieurs classes parentes implémentent la même méthode. Une bonne conception implique que chaque méthode doit avoir la même signature

lors de leur appels dans tous les cas (parce que l'ordre des appels est déterminée à l'exécution, parce que l'ordre s'adapte aux changements dans la hiérarchie, et parce que l'ordre peut inclure des classes sœurs inconnues avant l'exécution).

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

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

Notez que `super()` fait partie de l'implémentation du processus de liaison de recherche d'attributs pointés explicitement tel que `super().__getitem__(name)`. Il le fait en implémentant sa propre méthode `__getattr__()` pour rechercher les classes dans un ordre prévisible supportant l'héritage multiple coopératif. En conséquence, `super()` n'est pas défini pour les recherches implicites via des instructions ou des opérateurs tel que `super()[name]`.

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

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

tuple (*[iterable]*)

Plutôt qu'être une fonction, `tuple` est en fait un type de séquence immuable, tel que documenté dans [Tuples et Types séquentiels — list, tuple, range](#).

class type (*object*)

class type (*name, bases, dict*)

Avec un argument, donne le type d'*object*. La valeur donnée est un objet type et généralement la même que la valeur de l'attribut `object.__class__`.

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

Avec trois arguments, renvoie un nouveau type. C'est essentiellement une forme dynamique de l'instruction `class`. La chaîne *name* est le nom de la classe et deviendra l'attribut `__name__`; le *tuple bases* contient les classes mères et deviendra l'attribut `__bases__`; et le dictionnaire *dict* est l'espace de nommage contenant les définitions du corps de la classe, il est copié vers un dictionnaire standard pour devenir l'attribut `__dict__`. Par exemple, les deux instructions suivantes créent deux instances identiques de `type` :

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

Voir aussi [Objets type](#).

Modifié dans la version 3.6 : Les sous-classes de `type` qui ne redéfinissent pas `type.__new__` ne devraient plus utiliser la forme à un argument pour récupérer le type d'un objet.

vars (*[object]*)

Renvoie l'attribut `__dict__` d'un module, d'une classe, d'une instance ou de n'importe quel objet avec un attribut `__dict__`.

Les objets tels que les modules et les instances ont un attribut `__dict__` modifiable; Cependant, d'autres objets peuvent avoir des restrictions en écriture sur leurs attributs `__dict__` (par exemple, les classes utilisent un `types.MappingProxyType` pour éviter les modifications directes du dictionnaire).

Sans argument, `vars()` se comporte comme `locals()`. Notez que le dictionnaire des variables locales n'est utile qu'en lecture, car ses écritures sont ignorées.

zip (*iterables)

Construit un itérateur agréant les éléments de tous les itérables.

Donne un itérateur de tuples, où le *i*-ième tuple contient le *i*-ième élément de chacune des séquences ou itérables fournis. L'itérateur s'arrête lorsque le plus petit itérable fourni est épuisé. Avec un seul argument itérable, elle donne un itérateur sur des *tuples* d'un élément. Sans arguments, elle donne un itérateur vide. Équivalent à :

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

Il est garanti que les itérables soient évalués de gauche à droite. Cela rend possible de grouper une séquence de données en groupes de taille *n* via `zip(*[iter(s)]*n)`. Cela duplique le *même* itérateur *n* fois tel que le tuple obtenu contient le résultat de *n* appels à l'itérateur. Cela a pour effet de diviser la séquence en morceaux de taille *n*. `zip()` ne devrait être utilisée avec des itérables de longueur différente que lorsque les dernières données des itérables les plus longs peuvent être ignorées. Si ces valeurs sont importantes, utilisez plutôt `itertools.zip_longest()`.

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

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

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

Note : C'est une fonction avancée qui n'est pas fréquemment nécessaire, contrairement à `importlib.import_module()`.

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

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

level permet de choisir entre importation absolue ou relative. 0 (par défaut) implique de n'effectuer que des importations absolues. Une valeur positive indique le nombre de dossiers parents relativement au dossier du module appelant `__import__()` (voir la [PEP 328](#)).

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

module nommé par *name* est donné.

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

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

L'instruction `import ham.ham` appelle :

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

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

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

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

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

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

Modifié dans la version 3.3 : Des valeurs négatives pour *level* ne sont plus gérées (ce qui change la valeur par défaut pour 0).

Notes

Constantes natives

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

False

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

True

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

None

`None` est l'unique valeur du type `NoneType`. Elle est utilisée fréquemment pour représenter l'absence de valeur, comme lorsque des arguments par défaut ne sont pas passés à une fonction. Les assignations à `None` ne sont pas autorisées et lèvent une *SyntaxError*.

NotImplemented

Valeur spéciale qui devrait être renvoyée par les méthodes magiques à deux opérandes (e.g. `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) pour indiquer que l'opération n'est pas implémentée pour l'autre type; peut être renvoyé par les méthodes magiques augmentées à deux opérandes (e.g. `__imul__()`, `__iand__()`, etc.) avec le même objectif. Sa valeur booléenne est `True`.

Note : Lorsqu'une méthode à deux opérandes renvoie `NotImplemented`, l'interpréteur essaye d'exécuter l'opération réfléchie sur l'autre type (il existe d'autres mécanismes de *fallback*, en fonction de l'opérateur). Si toutes les tentatives renvoient `NotImplemented`, l'interpréteur lève une exception appropriée. Renvoyer `NotImplemented` à tort donne lieu à un message d'erreur peu clair ou au renvoi de la valeur `NotImplemented` pour le code Python.

Voir *Implémentation des opérations arithmétiques* pour des exemples.

Note : `NotImplementedError` et `NotImplemented` ne sont pas interchangeables, même s'ils ont un nom et un objectif similaire. Voir *NotImplementedError* pour savoir quand l'utiliser.

Ellipsis

The same as the ellipsis literal « `...` ». Special value used mostly in conjunction with extended slicing syntax for user-defined container data types.

`__debug__`

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

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

3.1 Constantes ajoutées par le module `site`

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

`quit` (`code=None`)

`exit` (`code=None`)

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

`copyright`

`credits`

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

`license`

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

CHAPITRE 4

Types natifs

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

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

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

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

4.1 Valeurs booléennes

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

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

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

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

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

4.2 Opérations booléennes — and, or, not

Ce sont les opérations booléennes, classées par priorité ascendante :

Opération	Résultat	Notes
<code>x or y</code>	si <code>x</code> est faux, alors <code>y</code> , sinon <code>x</code>	(1)
<code>x and y</code>	si <code>x</code> est faux, alors <code>x</code> , sinon <code>y</code>	(2)
<code>not x</code>	si <code>x</code> est faux, alors <code>True</code> , sinon <code>False</code>	(3)

Notes :

- (1) Ceci est un opérateur court-circuit : il n'évalue le deuxième argument que si le premier est faux.
- (2) Ceci est un opérateur court-circuit, il n'évalue le deuxième argument si le premier est vrai.
- (3) `not` a une priorité inférieure à celle des opérateurs non-booléens, donc `not a == b` est interprété comme `not (a == b)` et `a == not b` est une erreur de syntaxe.

4.3 Comparaisons

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

Ce tableau résume les opérations de comparaison :

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

Les objets de différents types, à l'exception de différents types numériques, ne peuvent en aucun cas être égaux. En outre, certains types (par exemple, les objets fonction) ne gèrent qu'une notion dégénérée de la comparaison où deux objets de ce type sont inégaux. Les opérateurs `<`, `<=`, `>` et `>=` lèvent une exception `TypeError` lorsqu'on compare un nombre complexe avec un autre type natif numérique, lorsque les objets sont de différents types qui ne peuvent pas être comparés, ou dans d'autres cas où il n'y a pas d'ordre défini.

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

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

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

Two more operations with the same syntactic priority, `in` and `not in`, are supported by types that are *iterable* or implement the `__contains__()` method.

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

Il existe trois types numériques distincts : *integers* (entiers), *floating point numbers* (nombres flottants) et *complex numbers* (nombres complexes). En outre, les booléens sont un sous-type des entiers. Les entiers ont une précision illimitée. Les nombres à virgule flottante sont généralement implémentés en utilisant des `double` en C ; des informations sur la précision et la représentation interne des nombres à virgule flottante pour la machine sur laquelle le programme est en cours d'exécution est disponible dans `sys.float_info`. Les nombres complexes ont une partie réelle et une partie imaginaire, qui sont chacune des nombres à virgule flottante. Pour extraire ces parties d'un nombre complexe `z`, utilisez `z.real` et `z.imag`. (La bibliothèque standard comprend d'autres types numériques, *fractions* qui stocke des rationnels et *decimal* qui stocke les nombres à virgule flottante avec une précision définissable par l'utilisateur.)

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

Python gère pleinement l'arithmétique mixte : quand un opérateur arithmétique binaire a des opérandes de types numériques différents, l'opérande avec le type « le plus étroit » est élargie au type de l'autre, où l'entier est plus étroit que la virgule flottante, qui est plus étroite que les complexes. Les comparaisons entre des nombres de type mixte utilisent la même règle.² Les constructeurs `int()`, `float()` et `complex()` peuvent être utilisés pour produire des nombres d'un type spécifique.

Tous les types numériques (sauf complexe) gèrent les opérations suivantes, classées par priorité ascendante (toutes les opérations numériques ont une priorité plus élevée que les opérations de comparaison) :

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

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

Notes :

- (1) Également appelé division entière. La valeur résultante est un nombre entier, bien que le type du résultat ne soit pas nécessairement *int*. Le résultat est toujours arrondi vers moins l'infini : $1//2$ vaut 0, $(-1)//2$ vaut -1, $1//(-2)$ vaut -1, et $(-1)//(-2)$ vaut 0.
- (2) Pas pour les nombres complexes. Convertissez-les plutôt en nombres flottants à l'aide de *abs()* si c'est approprié.
- (3) La conversion de virgule flottante en entier peut arrondir ou tronquer comme en C ; voir les fonctions *math.floor()* et *math.ceil()* pour des conversions bien définies.
- (4) *float* accepte aussi les chaînes *nan* et *inf* avec un préfixe optionnel + ou - pour *Not a Number* (NaN) et les infinis positif ou négatif.
- (5) Python définit *pow(0, 0)* et $0 ** 0$ valant 1, puisque c'est courant pour les langages de programmation, et logique.
- (6) Les littéraux numériques acceptés comprennent les chiffres 0 à 9 ou tout équivalent Unicode (caractères avec la propriété Nd).
Voir <http://www.unicode.org/Public/9.0.0/ucd/extracted/DerivedNumericType.txt> pour une liste complète des caractères avec la propriété Nd.

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

Opération	Résultat
<i>math.trunc(x)</i>	<i>x</i> tronqué à l' <i>Integral</i>
<i>round(x[, n])</i>	<i>x</i> arrondi à <i>n</i> chiffres, arrondissant la moitié au pair. Si <i>n</i> est omis, la valeur par défaut à 0.
<i>math.floor(x)</i>	le plus grand <i>Integral</i> $\leq x$
<i>math.ceil(x)</i>	le plus petit <i>Integral</i> $\geq x$

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

4.4.1 Opérations sur les bits des nombres entiers

Bitwise operations only make sense for integers. The result of bitwise operations is calculated as though carried out in two's complement with an infinite number of sign bits.

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

Ce tableau répertorie les opérations binaires triées par priorité ascendante :

Opération	Résultat	Notes
$x \mid y$	<i>ou <or></i> binaire de <i>x</i> et <i>y</i>	(4)
$x \wedge y$	<i>ou <or></i> exclusive binaire de <i>x</i> et <i>y</i>	(4)
$x \& y$	<i>et binaire <and></i> de <i>x</i> et <i>y</i>	(4)
$x \ll n$	<i>x</i> décalé vers la gauche de <i>n</i> bits	(1)(2)
$x \gg n$	<i>x</i> décalé vers la droite de <i>n</i> bits	(1)(3)
$\sim x$	les bits de <i>x</i> , inversés	

Notes :

- (1) Des valeurs de décalage négatives sont illégales et provoquent une exception *ValueError*.
- (2) Un décalage à gauche de *n* bits est équivalent à la multiplication par *pow(2, n)* sans vérification de débordement.
- (3) Un décalage à droite de *n* les bits est équivalent à la division par *pow(2, n)* sans vérification de débordement.

- (4) Performing these calculations with at least one extra sign extension bit in a finite two's complement representation (a working bit-width of $1 + \max(x.\text{bit_length}(), y.\text{bit_length}())$ or more) is sufficient to get the same result as if there were an infinite number of sign bits.

4.4.2 Méthodes supplémentaires sur les entiers

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

`int.bit_length()`

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

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

Plus précisément, si x est différent de zéro, `x.bit_length()` est le nombre entier positif unique, k tel que $2^{k-1} \leq \text{abs}(x) < 2^k$. Équivalentement, quand $\text{abs}(x)$ est assez petit pour avoir un logarithme correctement arrondi, $k = 1 + \text{int}(\log(\text{abs}(x), 2))$. Si x est nul, alors `x.bit_length()` donne 0. Équivalent à :

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

Nouveau dans la version 3.1.

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

Renvoie un tableau d'octets représentant un nombre entier.

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

L'entier est représenté par `length` octets. Une exception `OverflowError` est levée s'il n'est pas possible de représenter l'entier avec le nombre d'octets donnés.

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

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

Nouveau dans la version 3.2.

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

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

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

L'argument *bytes* doit être soit un *bytes-like object* soit un itérable produisant des *bytes*.

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

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

Nouveau dans la version 3.2.

4.4.3 Méthodes supplémentaires sur les nombres à virgule flottante

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

float.as_integer_ratio()

Renvoie une paire de nombres entiers dont le rapport est exactement égal au nombre d'origine et avec un dénominateur positif. Lève *OverflowError* avec un infini et *ValueError* avec un NaN.

float.is_integer()

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

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

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

float.hex()

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

classmethod *float.fromhex(s)*

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

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

Une chaîne hexadécimale prend la forme :

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

où *sign* peut être soit + soit -, *integer* et *fraction* sont des chaînes de chiffres hexadécimales, et *exponent* est un entier décimal facultativement signé. La casse n'est pas significative, et il doit y avoir au moins un chiffre hexadécimal soit dans le nombre entier soit dans la fraction. Cette syntaxe est similaire à la syntaxe spécifiée dans la section 6.4.4.2 de

la norme C99, et est aussi la syntaxe utilisée à partir de Java 1.5. En particulier, la sortie de `float.hex()` est utilisable comme valeur hexadécimale à virgule flottante littérale en C ou Java, et des chaînes hexadécimales produites en C via un format `%a` ou Java via `Double.toHexString` sont acceptées par `float.fromhex()`.

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

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

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

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

4.4.4 Hachage des types numériques

Pour deux nombres égaux `x` et `y` (`x == y`), pouvant être de différents types, il est requis que `hash(x) == hash(y)` (voir la documentation de `__hash__()`). Pour faciliter la mise en œuvre et l'efficacité à travers une variété de types numériques (y compris `int`, `float`, `decimal.Decimal` et `fractions.Fraction`) le hachage en Python pour les types numérique est basé sur une fonction mathématique unique qui est définie pour tout nombre rationnel, et donc s'applique à toutes les instances de `int` et `fractions.Fraction`, et toutes les instances finies de `float` et `decimal.Decimal`. Essentiellement, cette fonction est donnée par la réduction modulo `P` pour un nombre `P` premier fixe. La valeur de `P` est disponible comme attribut `modulus` de `sys.hash_info`.

CPython implementation detail : Actuellement, le premier utilisé est $P = 2^{31} - 1$ sur des machines dont les *longs* en C sont de 32 bits $P = 2^{61} - 1$ sur des machines dont les *longs* en C font 64 bits.

Voici les règles en détail :

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

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

```
import sys, math

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

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

(suite sur la page suivante)

```

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

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

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

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

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

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

```

4.5 Les types itérateurs

Python supporte un concept d'itération sur les conteneurs. C'est implémenté en utilisant deux méthodes distinctes qui permettent aux classes définies par l'utilisateur de devenir itérables. Les séquences, décrites plus bas en détail, supportent toujours les méthodes d'itération.

Une méthode doit être définie afin que les objets conteneurs supportent l'itération :

`container.__iter__()`

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

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

<iterator protocol> :

`iterator.__iter__()`

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

`iterator.__next__()`

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

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

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

4.5.1 Types générateurs

Les *generators* offrent un moyen pratique d'implémenter le protocole d'itération. Si la méthode `__iter__()` d'un objet conteneur est implémentée comme un générateur, elle renverra automatiquement un objet *iterator* (techniquement, un objet générateur) fournissant les méthodes `__iter__()` et `__next__()`. Plus d'informations sur les générateurs peuvent être trouvés dans la documentation de l'expression `yield`.

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

Il existe trois types séquentiels basiques : les *lists*, *tuples* et les *range*. D'autres types séquentiels spécifiques au traitement de *données binaires* et *chaînes de caractères* sont décrits dans des sections dédiées.

4.6.1 Opérations communes sur les séquences

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

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

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

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

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

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

Notes :

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

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

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

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

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

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

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

- (3) Si *i* ou *j* sont négatifs, l'indice est relatif à la fin de la séquence *s* : `len(s) + i` ou `len(s) + j` est substitué. Mais notez que `-0` est toujours 0.
- (4) La tranche de *s* de *i* à *j* est définie comme la séquence d'éléments d'indice *k* tels que `i <= k < j`. Si *i* ou *j* est supérieur à `len(s)`, `len(s)` est utilisé. Si *i* est omis ou None, 0 est utilisé. Si *j* est omis ou None, `len(s)` est utilisé. Si *i* est supérieure ou égale à *j*, la tranche est vide.

- (5) La tranche de `s` de `i` à `j` avec un pas de `k` est définie comme la séquence d'éléments d'indice $x = i + n * k$ tels que $0 \leq n < (j-i)/k$. En d'autres termes, les indices sont `i`, `i+k`, `i+2*k`, `i+3*k` et ainsi de suite, en arrêtant lorsque `j` est atteint (mais jamais inclus). Si `k` est positif, `i` et `j` sont réduits, s'ils sont plus grands, à `len(s)`. Si `k` est négatif, `i` et `j` sont réduits à `len(s) - 1` s'ils sont plus grands. Si `i` ou `j` sont omis ou sont `None`, ils deviennent des valeurs « extrêmes » (où l'ordre dépend du signe de `k`). Remarquez, `k` ne peut pas valoir zéro. Si `k` est `None`, il est traité comme 1.
- (6) Concaténer des séquences immuables donne toujours un nouvel objet. Cela signifie que la construction d'une séquence par concaténations répétées aura une durée d'exécution quadratique par rapport à la longueur de la séquence totale. Pour obtenir un temps d'exécution linéaire, vous devez utiliser l'une des alternatives suivantes :
- si vous concaténez des `str`, vous pouvez construire une liste puis utiliser `str.join()` à la fin, ou bien écrire dans une instance de `io.StringIO` et récupérer sa valeur lorsque vous avez terminé
 - si vous concaténez des `bytes`, vous pouvez aussi utiliser `bytes.join()` ou `io.BytesIO`, ou vous pouvez faire les concaténation sur place avec un objet `bytearray`. Les objets `bytearray` sont muables et ont un mécanisme de sur-allocation efficace
 - si vous concaténez des `tuple`, utilisez plutôt `extend` sur une `list`
 - pour d'autres types, cherchez dans la documentation de la classe concernée
- (7) Certains types séquentiels (tels que `range`) ne supportent que des séquences qui suivent des modèles spécifiques, et donc ne prennent pas en charge la concaténation ou la répétition.
- (8) `index` lève une exception `ValueError` quand `x` ne se trouve pas dans `s`. Toutes les implémentations ne gèrent pas les deux paramètres supplémentaires `i` et `j`. Ces deux arguments permettent de chercher efficacement dans une sous-séquence de la séquence. Donner ces arguments est plus ou moins équivalent à `s[i:j].index(x)`, sans copier les données ; l'indice renvoyé alors relatif au début de la séquence plutôt qu'au début de la tranche.

4.6.2 Types de séquences immuables

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

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

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

4.6.3 Types de séquences muables

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

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

Opération	Résultat	Notes
<code>s[i] = x</code>	élément <i>i</i> de <i>s</i> est remplacé par <i>x</i>	
<code>s[i:j] = t</code>	tranche de <i>s</i> de <i>i</i> à <i>j</i> est remplacée par le contenu de l'itérable <i>t</i>	
<code>del s[i:j]</code>	identique à <code>s[i:j] = []</code>	
<code>s[i:j:k] = t</code>	les éléments de <code>s[i:j:k]</code> sont remplacés par ceux de <i>t</i>	(1)
<code>del s[i:j:k]</code>	supprime les éléments de <code>s[i:j:k]</code> de la liste	
<code>s.append(x)</code>	ajoute <i>x</i> à la fin de la séquence (identique à <code>s[len(s):len(s)] = [x]</code>)	
<code>s.clear()</code>	removes all items from <i>s</i> (same as <code>del s[:]</code>)	(5)
<code>s.copy()</code>	creates a shallow copy of <i>s</i> (same as <code>s[:]</code>)	(5)
<code>s.extend(t)</code> or <code>s += t</code>	étend <i>s</i> avec le contenu de <i>t</i> (proche de <code>s[len(s):len(s)] = t</code>)	
<code>s *= n</code>	met à jour <i>s</i> avec son contenu répété <i>n</i> fois	(6)
<code>s.insert(i, x)</code>	insère <i>x</i> dans <i>s</i> à l'index donné par <i>i</i> (identique à <code>s[i:i] = [x]</code>)	
<code>s.pop([i])</code>	récupère l'élément à <i>i</i> et le supprime de <i>s</i>	(2)
<code>s.remove(x)</code>	supprime le premier élément de <i>s</i> pour qui <code>s[i] == x</code>	(3)
<code>s.reverse()</code>	inverse sur place les éléments de <i>s</i>	(4)

Notes :

- (1) *t* doit avoir la même longueur que la tranche qu'il remplace.
- (2) L'argument optionnel *i* vaut `-1` par défaut, afin que, par défaut, le dernier élément soit retiré et renvoyé.
- (3) `remove` lève une exception `ValueError` si *x* ne se trouve pas dans *s*.
- (4) La méthode `reverse()` modifie les séquence sur place pour économiser de l'espace lors du traitement de grandes séquences. Pour rappeler aux utilisateurs qu'elle a un effet de bord, elle ne renvoie pas la séquence inversée.
- (5) `clear()` et `copy()` sont incluses pour la compatibilité avec les interfaces des conteneurs muables qui ne supportent pas les opérations de découpage (comme `dict` et `set`)
Nouveau dans la version 3.3 : méthodes `clear()` et `copy()`.
- (6) La valeur *n* est un entier, ou un objet implémentant `__index__()`. Zéro et les valeurs négatives de *n* permettent d'effacer la séquence. Les éléments dans la séquence ne sont pas copiés ; ils sont référencés plusieurs fois, comme expliqué pour `s * n` dans *Opérations communes sur les séquences*.

4.6.4 Listes

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

class `list` (`[iterable]`)

Les listes peuvent être construites de différentes manières :

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

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

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

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

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

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

`sort()` accepte deux arguments qui ne peuvent être fournis que par mot-clé (*keyword-only arguments*) :

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

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

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

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

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

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

4.6.5 Tuples

Les tuples (*n-uplets* en français) sont des séquences immuables, généralement utilisées pour stocker des collections de données hétérogènes (tels que les tuples de deux éléments produits par la fonction native `enumerate()`). Les tuples sont également utilisés dans des cas où une séquence homogène et immuable de données est nécessaire (pour, par exemple, les stocker dans un `set` ou un `dict`).

class tuple ([*iterable*])

Les tuples peuvent être construits de différentes façons :

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

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

Notez que c'est en fait la virgule qui fait un tuple et non les parenthèses. Les parenthèses sont facultatives, sauf dans le cas du tuple vide, ou lorsqu'elles sont nécessaires pour éviter l'ambiguïté syntaxique. Par exemple, `f(a, b, c)` est un appel de fonction avec trois arguments, alors que `f((a, b, c))` est un appel de fonction avec un tuple de trois éléments comme unique argument.

Les tuples implémentent toutes les opérations *communes* des séquences.

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

4.6.6 Ranges

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

class `range` (*stop*)

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

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

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

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

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

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

Exemples avec `range` :

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

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

start

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

stop

La valeur du paramètre `stop`

step

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

L'avantage du type `range` sur une `list` classique ou `tuple` est qu'un objet `range` prendra toujours la même (petite) quantité de mémoire, peu importe la taille de la gamme qu'elle représente (car elle ne stocke que les valeurs `start`, `stop` et `step`, le calcul des éléments individuels et les sous-intervalles au besoin).

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

```
>>> r = range(0, 20, 2)
>>> r
```

(suite sur la page suivante)

(suite de la page précédente)

```

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

```

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

Modifié dans la version 3.2 : Implémente la classe de base abstraite *Sequence*. Supporte le *slicing* et les indices négatifs. Tester l'appartenance d'un *int* en temps constant au lieu d'itérer tous les éléments.

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

Nouveau dans la version 3.3 : Les attributs *start*, *stop* et *step*.

Voir aussi :

- The [linspace recipe](#) shows how to implement a lazy version of range suitable for floating point applications.

4.7 Type Séquence de Texte — `str`

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

- Les guillemets simples : `'autorisent les "guillemets"'`
- Les guillemets : `"autorisent les guillemets 'simples'"`.
- Guillemets triples : `'''Trois guillemets simples'''`, `"""Trois guillemets"""`

Les chaînes entre triple guillemets peuvent couvrir plusieurs lignes, tous les espaces associés seront inclus dans la chaîne littérale.

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

Voir *strings* pour plus d'informations sur les différentes formes de chaînes littérales, y compris des séquences d'échappement prises en charge, et le préfixe *r* (*raw* (brut)) qui désactive la plupart des traitements de séquence d'échappement.

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

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

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

Modifié dans la version 3.3 : Pour une compatibilité ascendante avec la série Python 2, le préfixe *u* est à nouveau autorisé sur les chaînes littérales. Elle n'a aucun effet sur le sens des chaînes littérales et ne peut être combiné avec le préfixe *r*.


```
class str(object="")
```

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

Renvoie une représentation *string* de *object*. Si *object* n'est pas fourni, renvoie une chaîne vide. Sinon, le comportement de `str()` dépend de si *encoding* ou *errors* sont donnés, voir l'exemple.

Si ni *encoding* ni *errors* ne sont donnés, `str(object)` renvoie `object.__str__()`, qui est la représentation de chaîne « informelle » ou bien affichable de *object*. Pour les chaînes, c'est la chaîne elle-même. Si *object* n'a pas de méthode `__str__()`, `str()` utilise `repr(object)`.

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

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

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

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

4.7.1 Méthodes de chaînes de caractères

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

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

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

`str.capitalize()`

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

`str.casefold()`

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

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

L'algorithme de *casefolding* est décrit dans la section 3.13 de la norme Unicode.

Nouveau dans la version 3.3.

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

Donne la chaîne au centre d'une chaîne de longueur *width*. Le remplissage est fait en utilisant l'argument *fillchar* (qui par défaut est un espace ASCII). La chaîne d'origine est renvoyée si *width* est inférieur ou égale à `len(s)`.

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

Donne le nombre d'occurrences de *sub* ne se chevauchant pas dans la *range* `[start, end]`. Les arguments facultatifs *start* et *end* sont interprétés comme pour des *slices*.

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

Donne une version encodée de la chaîne sous la forme d'un objet *bytes*. L'encodage par défaut est 'utf-8'.

`errors` peut être donné pour choisir un autre système de gestion d'erreur. La valeur par défaut pour `errors` est `'strict'`, ce qui signifie que les erreurs d'encodage lèvent une `UnicodeError`. Les autres valeurs possibles sont `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` et tout autre nom enregistré via `codecs.register_error()`, voir la section *Gestionnaires d'erreurs*. Pour une liste des encodages possibles, voir la section *Standard Encodings*.

Modifié dans la version 3.1 : Gestion des arguments par mot clef.

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

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

`str.expandtabs(tabsize=8)`

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

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

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

Donne la première la position dans la chaîne où `sub` est trouvé dans le *slice* `s[start:end]`. Les arguments facultatifs `start` et `end` sont interprétés comme dans la notation des *slice*. Donne `-1` si `sub` n'est pas trouvé.

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

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

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

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

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

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

Note : When formatting a number (`int`, `float`, `complex`, `decimal.Decimal` and subclasses) with the `n` type (ex: `'{:n}'.format(1234)`), the function temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale to decode `decimal_point` and `thousands_sep` fields of `localeconv()` if they are non-ASCII or longer than 1 byte, and the `LC_NUMERIC` locale is different than the `LC_CTYPE` locale. This temporary change affects other threads.

Modifié dans la version 3.6.5 : Lors du formatage d'un nombre avec le format `n`, la fonction change temporairement `LC_CTYPE` par la valeur de `LC_NUMERIC` dans certains cas.

`str.format_map(mapping)`

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

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

Nouveau dans la version 3.2.

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

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

`str.isalnum()`

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

`str.isalpha()`

Donne `True` si tous les caractères de la chaîne sont alphabétiques et qu'elle contient au moins un caractère, sinon `False`. Les caractères alphabétiques sont les caractères définis dans la base de données de caractères Unicode comme *Letter*, à savoir, ceux qui ont « Lm », « Lt », « Lu », « Ll », ou « Lo » comme catégorie générale. Notez que ceci est différent de la propriété *Alphabetic* définie dans la norme Unicode.

`str.isdecimal()`

Renvoie vrai si tous les caractères de la chaîne sont des caractères décimaux et qu'elle contient au moins un caractère, sinon faux. Les caractères décimaux sont ceux pouvant être utilisés pour former des nombres en base 10, tels que U+0660, ARABIC-INDIC DIGIT ZERO. Spécifiquement, un caractère décimal est un caractère dans la catégorie Unicode générale « Nd ».

`str.isdigit()`

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

`str.isidentifier()`

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

Utilisez `keyword.iskeyword()` pour savoir si un identifiant est réservé, tels que `def` et `class`.

`str.islower()`

Donne `True` si tous les caractères capitalisables⁴ de la chaîne sont en minuscules et qu'elle contient au moins un caractère capitalisable. Donne `False` dans le cas contraire.

`str.isnumeric()`

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

4. Les caractères capitalisables sont ceux dont la propriété Unicode *general category* est soit « Lu » (pour *Letter, uppercase*), soit « Ll » (pour *Letter, lowercase*), soit « Lt » (pour *Letter, titlecase*).

`str.isprintable()`

Donne `True` si tous les caractères de la chaîne sont affichables ou qu'elle est vide sinon, `False`. Les caractères non affichables sont les caractères définis dans la base de données de caractères Unicode comme « *Other* » ou « *Separator* », à l'exception de l'espace ASCII (0x20) qui est considéré comme affichable. (Notez que les caractères imprimables dans ce contexte sont ceux qui ne devraient pas être protégés quand `repr()` est invoquée sur une chaîne. Ça n'a aucune incidence sur le traitement des chaînes écrites sur `sys.stdout` ou `sys.stderr`.)

`str.isspace()`

Donne `True` s'il n'y a que des caractères blancs dans la chaîne et il y a au moins un caractère, sinon `False`. Les caractères blancs sont les caractères définis dans la base de données de caractères Unicode comme « *Other* » ou « *Separator* » ainsi que ceux ayant la propriété bidirectionnelle valant « WS », « B » ou « S ».

`str.istitle()`

Donne `True` si la chaîne est une chaîne *titlecased* et qu'elle contient au moins un caractère, par exemple les caractères majuscules ne peuvent suivre les caractères non capitalisables et les caractères minuscules ne peuvent suivre que des caractères capitalisables. Donne `False` dans le cas contraire.

`str.isupper()`

Donne `True` si tous les caractères différenciables sur la casse⁴ de la chaîne sont en majuscules et il y a au moins un caractère différenciable sur la casse, sinon `False`.

`str.join(iterable)`

Donne une chaîne qui est la concaténation des chaînes contenues dans *iterable*. Une `TypeError` sera levée si une valeur d'*iterable* n'est pas une chaîne, y compris pour les objets *bytes*. Le séparateur entre les éléments est la chaîne fournissant cette méthode.

`str.ljust(width[, fillchar])`

Renvoie la chaîne justifiée à gauche dans une chaîne de longueur *width*. Le rembourrage est fait en utilisant *fillchar* (qui par défaut est un espace ASCII). La chaîne d'origine est renvoyée si *width* est inférieur ou égale à `len(s)`.

`str.lower()`

Renvoie une copie de la chaîne avec tous les caractères capitalisables⁴ convertis en minuscules. L'algorithme de mise en minuscules utilisé est décrit dans la section 3.13 de la norme Unicode.

`str.lstrip([chars])`

Renvoie une copie de la chaîne des caractères supprimés au début. L'argument *chars* est une chaîne spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, la valeur par défaut de *chars* permet de supprimer des espaces. L'argument *chars* n'est pas un préfixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> '   spacious   '.lstrip()
'spacious'
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

`static str.maketrans(x[, y[, z]])`

Cette méthode statique renvoie une table de traduction utilisable pour `str.translate()`.

Si un seul argument est fourni, ce soit être un dictionnaire faisant correspondre des points de code Unicode (nombres entiers) ou des caractères (chaînes de longueur 1) à des points de code Unicode.

Si deux arguments sont fournis, ce doit être deux chaînes de caractères de même longueur. Le dictionnaire renvoyé fera correspondre pour chaque caractère de *x* un caractère de *y* pris à la même place. Si un troisième argument est fourni, ce doit être une chaîne dont chaque caractère correspondra à `None` dans le résultat.

`str.partition(sep)`

Divise la chaîne à la première occurrence de *sep*, et donne un *tuple* de trois éléments contenant la partie avant le séparateur, le séparateur lui-même, et la partie après le séparateur. Si le séparateur n'est pas trouvé, le *tuple* contiendra la chaîne elle-même, suivie de deux chaînes vides.

`str.replace(old, new[, count])`

Renvoie une copie de la chaîne dont toutes les occurrences de la sous-chaîne *old* sont remplacés par *new*. Si l'argument optionnel *count* est donné, seules les *count* premières occurrences sont remplacées.

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

Donne l'indice le plus élevé dans la chaîne où la sous-chaîne *sub* se trouve, de telle sorte que *sub* soit contenue dans `s[start:end]`. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des *slices*. Donne `-1` en cas d'échec.

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

Comme `rfind()` mais lève une exception `ValueError` lorsque la sous-chaîne *sub* est introuvable.

`str.rjust(width[, fillchar])`

Renvoie la chaîne justifié à droite dans une chaîne de longueur *width*. Le rembourrage est fait en utilisant le caractère spécifié par *fillchar* (par défaut est un espace ASCII). La chaîne d'origine est renvoyée si *width* est inférieure ou égale à `len(s)`.

`str.rpartition(sep)`

Divise la chaîne à la dernière occurrence de *sep*, et donne un tuple de trois éléments contenant la partie avant le séparateur, le séparateur lui-même, et la partie après le séparateur. Si le séparateur n'est pas trouvé, le *tuple* contiendra deux chaînes vides, puis par la chaîne elle-même.

`str.rsplit(sep=None, maxsplit=-1)`

Renvoie une liste des mots de la chaîne, en utilisant *sep* comme séparateur. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être faites, celles « à droite ». Si *sep* est pas spécifié ou est `None`, tout espace est un séparateur. En dehors du fait qu'il découpe par la droite, `rsplit()` se comporte comme `split()` qui est décrit en détail ci-dessous.

`str.rstrip([chars])`

Renvoie une copie de la chaîne avec des caractères finaux supprimés. L'argument *chars* est une chaîne spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, les espaces sont supprimés. L'argument *chars* n'est pas un suffixe : toutes les combinaisons de ses valeurs sont retirées :

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

`str.split(sep=None, maxsplit=-1)`

Renvoie une liste des mots de la chaîne, en utilisant *sep* comme séparateur de mots. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être effectuées, (donnant ainsi une liste de longueur `maxsplit+1`). Si *maxsplit* n'est pas fourni, ou vaut `-1`, le nombre de découpes n'est pas limité (Toutes les découpes possibles sont faites).

Si *sep* est donné, les délimiteurs consécutifs ne sont pas regroupés et ainsi délimitent des chaînes vides (par exemple, `'1,,2'.split(',')` donne `['1', '', '2']`). L'argument *sep* peut contenir plusieurs caractères (par exemple, `'1<>2<>3'.split('<>')` renvoie `['1', '2', '3']`). Découper une chaîne vide en spécifiant *sep* donne `['']`.

Par exemple :

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3'.split(',')
['1', '2', '', '3', '']
```

Si *sep* n'est pas spécifié ou est `None`, un autre algorithme de découpage est appliqué : les espaces consécutifs sont considérés comme un seul séparateur, et le résultat ne contiendra pas les chaînes vides de début ou de la fin si la

chaîne est préfixée ou suffixée d'espaces. Par conséquent, diviser une chaîne vide ou une chaîne composée d'espaces avec un séparateur `None` renvoie `[]`.

Par exemple :

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> ' 1 2 3 '.split()
['1', '2', '3']
```

`str.splitlines([keepends])`

Renvoie les lignes de la chaîne sous forme de liste, la découpe se fait au niveau des limites des lignes. Les sauts de ligne ne sont pas inclus dans la liste des résultats, sauf si *keepends* est donné, et est vrai.

Cette méthode découpe sur les limites de ligne suivantes. Ces limites sont un sur ensemble de *universal newlines*.

Représentation	Description
<code>\n</code>	Saut de ligne
<code>\r</code>	Retour chariot
<code>\r\n</code>	Retour chariot + saut de ligne
<code>\v</code> or <code>\x0b</code>	Tabulation verticale
<code>\f</code> or <code>\x0c</code>	Saut de page
<code>\x1c</code>	Séparateur de fichiers
<code>\x1d</code>	Séparateur de groupes
<code>\x1e</code>	Séparateur d'enregistrements
<code>\x85</code>	Ligne suivante (code de contrôle <i>CI</i>)
<code>\u2028</code>	Séparateur de ligne
<code>\u2029</code>	Séparateur de paragraphe

Modifié dans la version 3.2 : `\v` et `\f` ajoutés à la liste des limites de lignes.

Par exemple :

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Contrairement à `split()` lorsque *sep* est fourni, cette méthode renvoie une liste vide pour la chaîne vide, et un saut de ligne à la fin ne se traduit pas par une ligne supplémentaire :

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

À titre de comparaison, `split('\n')` donne :

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

`str.startswith(prefix[, start[, end]])`

Donne `True` si la chaîne commence par *prefix*, sinon `False`. *prefix* peut aussi être un tuple de préfixes à rechercher. Lorsque *start* est donné, la comparaison commence à cette position, et lorsque *end* est donné, la comparaison s'arrête à celle ci.

`str.strip([chars])`

Donne une copie de la chaîne dont des caractères initiaux et finaux sont supprimés. L'argument *chars* est une chaîne spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, les espaces sont supprimés. L'argument *chars* est pas un préfixe ni un suffixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

Les caractères de *char* sont retirés du début et de la fin de la chaîne. Les caractères sont retirés de la gauche jusqu'à atteindre un caractère ne figurant pas dans le jeu de caractères dans *chars*. La même opération à lieu par la droite. Par exemple :

```
>>> comment_string = '#..... Section 3.2.1 Issue #32 .....'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

`str.swapcase()`

Renvoie une copie de la chaîne dont les caractères majuscules sont convertis en minuscules et vice versa. Notez qu'il est pas nécessairement vrai que `s.swapcase().swapcase() == s`.

`str.title()`

Renvoie une version en initiales majuscules de la chaîne où les mots commencent par une capitale et les caractères restants sont en minuscules.

Par exemple :

```
>>> 'Hello world'.title()
'Hello World'
```

Pour l'algorithme, la notion de mot est définie simplement et indépendamment de la langue comme un groupe de lettres consécutives. La définition fonctionne dans de nombreux contextes, mais cela signifie que les apostrophes (typiquement de la forme possessive en Anglais) forment les limites de mot, ce qui n'est pas toujours le résultat souhaité :

```
>>> "they're bill's friends from the UK".title()
'They'Re Bill'S Friends From The Uk'
```

Une solution pour contourner le problème des apostrophes peut être obtenue en utilisant des expressions rationnelles :

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() +
...                               mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase("they're bill's friends.")
'They're Bill's Friends.'
```

`str.translate(table)`

Renvoie une copie de la chaîne dans laquelle chaque caractère a été changé selon la table de traduction donnée. La table doit être un objet qui implémente l'indexation via `__getitem__()`, typiquement un *mapping* ou une *sequence*. Pour un ordinal Unicode (un entier), la table peut donner soit un ordinal Unicode ou une chaîne pour faire correspondre un ou plusieurs caractère au caractère donné, soit `None` pour supprimer le caractère de la chaîne de renvoyée soit lever une exception *LookupError* pour ne pas changer le caractère.

Vous pouvez utiliser `str.maketrans()` pour créer une table de correspondances de caractères dans différents formats.

Voir aussi le module `codecs` pour une approche plus souple de changements de caractères par correspondance.

`str.upper()`

Return a copy of the string with all the cased characters⁴ converted to uppercase. Note that `s.upper().isupper()` might be `False` if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not « Lu » (Letter, uppercase), but e.g. « Lt » (Letter, titlecase).

L'algorithme de capitalisation utilisé est décrit dans la section 3.13 de la norme Unicode.

`str.zfill(width)`

Renvoie une copie de la chaîne remplie par la gauche du chiffre (le caractère ASCII) '0' pour faire une chaîne de longueur `width`. Un préfixe ('+' / '-') est permis par l'insertion du caractère de remplissage *après* le caractère désigne plutôt qu'avant. La chaîne d'origine est renvoyée si `width` est inférieur ou égale à `len(s)`.

Par exemple :

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.7.2 Formatage de chaînes à la `printf`

Note : Ces opérations de mise en forme contiennent des bizarreries menant à de nombreuses erreurs classiques (tel que ne pas réussir à afficher des *tuples* ou des dictionnaires correctement). Utiliser les formatted string literals ou la méthode `str.format()` aide à éviter ces erreurs. Ces alternatives offrent aussi une approche plus puissante, flexible, et extensible de la mise en forme.

Les objets `str` n'exposent qu'une opération : L'opérateur % (modulo). Aussi connu sous le nom d'opérateur de formatage, ou opérateur d'interpolation. Étant donné `format % values` (où `format` est une chaîne), les marqueurs % de `format` sont remplacés par zéro ou plusieurs éléments de `values`. L'effet est similaire à la fonction `sprintf()` du langage C.

Si `format` ne nécessite qu'un seul argument, `values` peut être un objet unique.⁵ Si `values` est un tuple, il doit contenir exactement le nombre d'éléments spécifiés par la chaîne de format, ou un seul objet de correspondances (*mapping object*, par exemple, un dictionnaire).

Un indicateur de conversion contient deux ou plusieurs caractères et comporte les éléments suivants, qui doivent apparaître dans cet ordre :

1. Le caractère '%', qui marque le début du marqueur.
2. La clé de correspondance (facultative), composée d'une suite de caractères entre parenthèse (par exemple, `(somename)`).
3. Des options de conversion, facultatives, qui affectent le résultat de certains types de conversion.
4. Largeur minimum (facultative). Si elle vaut '*' (astérisque), la largeur est lue de l'élément suivant du tuple `values`, et l'objet à convertir vient après la largeur de champ minimale et la précision facultative.
5. Précision (facultatif), donnée sous la forme d'un '.' (point) suivi de la précision. Si la précision est '*' (un astérisque), la précision est lue à partir de l'élément suivant du tuple `values` et la valeur à convertir vient ensuite.
6. Modificateur de longueur (facultatif).
7. Type de conversion.

5. Pour insérer un *tuple*, vous devez donc donner un *tuple* d'un seul élément, contenant le *tuple* à insérer.

Lorsque l'argument de droite est un dictionnaire (ou un autre type de *mapping*), les marqueurs dans la chaîne *doivent* inclure une clé présente dans le dictionnaire, écrite entre parenthèses, immédiatement après le caractère '%'. La clé indique quelle valeur du dictionnaire doit être formatée. Par exemple :

```
>>> print('%(language)s has %(number)03d quote types.' %
...        {'language': "Python", "number": 2})
Python has 002 quote types.
```

Dans ce cas, aucune * ne peuvent se trouver dans le format (car ces * nécessitent une liste (accès séquentiel) de paramètres).

Les caractères indicateurs de conversion sont :

Op-tion	Signification
'#'	La conversion utilisera la « forme alternative » (définie ci-dessous).
'0'	Les valeurs numériques converties seront complétée de zéros.
'-'	La valeur convertie est ajustée à gauche (remplace la conversion '0' si les deux sont données).
' '	(un espace) Un espace doit être laissé avant un nombre positif (ou chaîne vide) produite par la conversion d'une valeur signée.
'+'	Un caractère de signe ('+' ou '-') précède la valeur convertie (remplace le marqueur « espace »).

Un modificateur de longueur (h, l ou L) peut être présent, mais est ignoré car il est pas nécessaire pour Python, donc par exemple %ld est identique à %d.

Les types utilisables dans les conversion sont :

Conversion	Signification	Notes
'd'	Entier décimal signé.	
'i'	Entier décimal signé.	
'o'	Valeur octale signée.	(1)
'u'	Type obsolète — identique à 'd'.	(6)
'x'	Hexadécimal signé (en minuscules).	(2)
'X'	Hexadécimal signé (capitales).	(2)
'e'	Format exponentiel pour un <i>float</i> (minuscule).	(3)
'E'	Format exponentiel pour un <i>float</i> (en capitales).	(3)
'f'	Format décimal pour un <i>float</i> .	(3)
'F'	Format décimal pour un <i>float</i> .	(3)
'g'	Format <i>float</i> . Utilise le format exponentiel minuscules si l'exposant est inférieur à -4 ou pas plus petit que la précision, sinon le format décimal.	(4)
'G'	Format <i>float</i> . Utilise le format exponentiel en capitales si l'exposant est inférieur à -4 ou pas plus petit que la précision, sinon le format décimal.	(4)
'c'	Un seul caractère (accepte des entiers ou une chaîne d'un seul caractère).	
'r'	String (convertit n'importe quel objet Python avec <i>repr()</i>).	(5)
's'	String (convertit n'importe quel objet Python avec <i>str()</i>).	(5)
'a'	String (convertit n'importe quel objet Python en utilisant <i>ascii()</i>).	(5)
'%'	Aucun argument n'est converti, donne un caractère de '%' dans le résultat.	

Notes :

- (1) La forme alternative entraîne l'insertion d'un préfixe octal ('0o') avant le premier chiffre.
- (2) La forme alternative entraîne l'insertion d'un préfixe '0x' ou '0X' (respectivement pour les formats 'x' et 'X') avant le premier chiffre.

- (3) La forme alternative implique la présence d'un point décimal, même si aucun chiffre ne le suit.
La précision détermine le nombre de chiffres après la virgule, 6 par défaut.
- (4) La forme alternative implique la présence d'un point décimal et les zéros non significatifs sont conservés (ils ne le seraient pas autrement).
La précision détermine le nombre de chiffres significatifs avant et après la virgule. 6 par défaut.
- (5) Si la précision est *N*, la sortie est tronquée à *N* caractères.
- (6) Voir la [PEP 237](#).

Puisque les chaînes Python ont une longueur explicite, les conversions `%s` ne considèrent pas `'\0'` comme la fin de la chaîne.

Modifié dans la version 3.1 : Les conversions `%f` pour nombres dont la valeur absolue est supérieure à $1e50$ ne sont plus remplacés par des conversions `%g`.

4.8 Séquences Binaires — `bytes`, `bytearray`, `memoryview`

Les principaux types natifs pour manipuler des données binaires sont `bytes` et `bytearray`. Ils sont supportés par `memoryview` qui utilise le buffer protocol pour accéder à la mémoire d'autres objets binaires sans avoir besoin d'en faire une copie.

Le module `array` permet le stockage efficace de types basiques comme les entiers de 32 bits et les `float` double précision IEEE754.

4.8.1 Objets `bytes`

Les `bytes` sont des séquences immuables d'octets. Comme beaucoup de protocoles binaires utilisent l'ASCII, les objets `bytes` offrent plusieurs méthodes qui ne sont valables que lors de la manipulation de données ASCII et sont étroitement liés aux objets `str` dans bien d'autres aspects.

class `bytes` (`[source[, encoding[, errors]]]`)

Tout d'abord, la syntaxe des `bytes` littéraux est en grande partie la même que pour les chaînes littérales, en dehors du préfixe `b` :

- Les guillemets simples : `b'...` autorisent aussi les guillemets "doubles".
- Les guillemets doubles : `b"..."` permettent aussi les guillemets 'simples'.
- Les guillemets triples : `b'''...'''` 3 single quotes, `b"""..."""` 3 double quotes.

Seuls les caractères ASCII sont autorisés dans les littéraux de `bytes` (quel que soit l'encodage du code source déclaré). Toutes les valeurs au delà de 127 doivent être entrées dans littéraux de `bytes` en utilisant une séquence d'échappement appropriée.

Comme avec les chaînes littérales, les `bytes` littéraux peuvent également utiliser un préfixe `r` pour désactiver le traitement des séquences d'échappement. Voir `strings` pour plus d'informations sur les différentes formes littérales de `bytes`, y compris les séquences d'échappement supportées.

While bytes literals and representations are based on ASCII text, bytes objects actually behave like immutable sequences of integers, with each value in the sequence restricted such that $0 \leq x < 256$ (attempts to violate this restriction will trigger `ValueError`). This is done deliberately to emphasise that while many binary formats include ASCII based elements and can be usefully manipulated with some text-oriented algorithms, this is not generally the case for arbitrary binary data (blindly applying text processing algorithms to binary data formats that are not ASCII compatible will usually lead to data corruption).

En plus des formes littérales, des objets `bytes` peuvent être créés par de nombreux moyens :

- Un objet `bytes` rempli de zéros d'une longueur spécifiée : `bytes(10)`
- D'un itérable d'entiers : `bytes(range(20))`
- Copier des données binaires existantes via le *buffer protocol* : `bytes(obj)`

Voir aussi la fonction native *bytes*.

Puisque 2 chiffres hexadécimaux correspondent précisément à un seul octet, les nombres hexadécimaux sont un format couramment utilisé pour décrire les données binaires. Par conséquent, le type *bytes* a une méthode de classe pour lire des données dans ce format :

classmethod *fromhex* (*string*)

Cette méthode de la classe *bytes* renvoie un objet *bytes*, décodant la chaîne donnée. La chaîne doit contenir deux chiffres hexadécimaux par octet, les espaces ASCII sont ignorés.

```
>>> bytes.fromhex('2Ef0 F1f2 ')
b'\xf0\xf1\xf2'
```

Une fonction de conversion inverse existe pour transformer un objet *bytes* en sa représentation hexadécimale.

hex ()

Renvoie une chaîne contenant deux chiffres hexadécimaux pour chaque octet du *byte*.

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

Nouveau dans la version 3.5.

Comme les objets *bytes* sont des séquences d'entiers (semblables à un tuple), pour une instance de *bytes* *b*, *b*[0] sera un entier, tandis que *b*[0:1] sera un objet *bytes* de longueur 1. (Cela contraste avec les chaînes, où l'indexation et le *slicing* donne une chaîne de longueur 1)

La représentation des *bytes* utilise le format littéral (*b'...'*) car il est souvent plus utile que par exemple *bytes([46, 46, 46])*. Vous pouvez toujours convertir un *bytes* en liste d'entiers en utilisant *list(b)*.

Note : Pour les utilisateurs de Python 2.x : Dans la série 2.x de Python, une variété de conversions implicites entre les chaînes 8-bit (la chose la plus proche d'un type natif de données binaires offert par Python 2) et des chaînes Unicode étaient permises. C'était une solution de contournement, pour garder la rétro-compatibilité, considérant que Python ne prenait initialement en charge que le texte 8 bits, le texte Unicode est un ajout ultérieur. En Python 3.x, ces conversions implicites ont disparues, les conversions entre les données binaires et texte Unicode doivent être explicites, et les *bytes* sont toujours différents des chaînes.

4.8.2 Objets *bytearray*

Les objets *bytearray* sont l'équivalent muable des objets *bytes*.

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

Il n'y a pas de syntaxe littérale dédiée aux *bytearray*, ils sont toujours créés en appelant le constructeur :

- Créer une instance vide : *bytearray()*
- Créer une instance remplie de zéros d'une longueur donnée : *bytearray(10)*
- À partir d'un itérable d'entiers : *bytearray(range(20))*
- Copie des données binaires existantes via le *buffer protocol* : *bytearray(b'Hi!')*

Comme les *bytearray* sont muables, ils prennent en charge les opérations de séquence *mutables* en plus des opérations communes de *bytes* et *bytearray* décrites dans *Opérations sur les bytes et bytearray*.

Voir aussi la fonction native *bytearray*.

Puisque 2 chiffres hexadécimaux correspondent précisément à un octet, les nombres hexadécimaux sont un format couramment utilisé pour décrire les données binaires. Par conséquent, le type *bytearray* a une méthode de classe pour lire les données dans ce format :

classmethod *fromhex* (*string*)

Cette méthode de la classe *bytearray* renvoie un objet *bytearray*, décodant la chaîne donnée. La chaîne doit contenir deux chiffres hexadécimaux par octet, les espaces ASCII sont ignorés.

```
>>> bytearray.fromhex('2Ef0 F1f2 ')
bytearray(b'.\xf0\xf1\xf2')
```

Une fonction de conversion inverse existe pour transformer un objet *bytearray* en sa représentation hexadécimale.

hex()

Renvoie une chaîne contenant deux chiffres hexadécimaux pour chaque octet du *byte*.

```
>>> bytearray(b'.\xf0\xf1\xf2').hex()
'f0f1f2'
```

Nouveau dans la version 3.5.

Comme les *bytearray* sont des séquences d'entiers (semblables à une liste), pour un objet *bytearray* *b*, *b*[0] sera un entier, tandis que *b*[0:1] sera un objet *bytearray* de longueur 1. (Ceci contraste avec les chaînes de texte, où l'indexation et le *slicing* produit une chaîne de longueur 1)

La représentation des objets *bytearray* utilise le format littéral des *bytes* (*bytearray*(b'...')) car il est souvent plus utile que par exemple *bytearray*([46, 46, 46]). Vous pouvez toujours convertir un objet *bytearray* en une liste de nombres entiers en utilisant *list(b)*.

4.8.3 Opérations sur les *bytes* et *bytearray*

bytes et *bytearray* prennent en charge les opérations *communes* des séquences. Ils interagissent non seulement avec des opérandes de même type, mais aussi avec les *bytes-like object*. En raison de cette flexibilité, ils peuvent être mélangés librement dans des opérations sans provoquer d'erreurs. Cependant, le type du résultat peut dépendre de l'ordre des opérandes.

Note : Les méthodes sur les *bytes* et les *bytearray* n'acceptent pas les chaînes comme arguments, tout comme les méthodes sur les chaînes n'acceptent pas les *bytes* comme arguments. Par exemple, vous devez écrire :

```
a = "abc"
b = a.replace("a", "f")
```

et :

```
a = b"abc"
b = a.replace(b"a", b"f")
```

Quelques opérations de *bytes* et *bytearray* supposent l'utilisation de formats binaires compatibles ASCII, et donc doivent être évités lorsque vous travaillez avec des données binaires arbitraires. Ces restrictions sont couvertes ci-dessous.

Note : Utiliser ces opérations basées sur l'ASCII pour manipuler des données binaires qui ne sont pas au format ASCII peut les corrompre.

Les méthodes suivantes sur les *bytes* et *bytearray* peuvent être utilisées avec des données binaires arbitraires.

```
bytes.count(sub[, start[, end]])
bytearray.count(sub[, start[, end]])
```

Renvoie le nombre d'occurrences qui ne se chevauchent pas de la sous-séquence *sub* dans l'intervalle [*start*, *end*]. Les arguments facultatifs *start* et *end* sont interprétés comme pour un *slice*.

La sous-séquence à rechercher peut être un quelconque *bytes-like object* ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3 : Accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

```
bytes.decode(encoding="utf-8", errors="strict")
```

```
bytearray.decode(encoding="utf-8", errors="strict")
```

Décode les octets donnés, et le renvoie sous forme d'une chaîne de caractères. L'encodage par défaut est 'utf-8'. *errors* peut être donné pour changer de système de gestion des erreurs. Sa valeur par défaut est 'strict', ce qui signifie que les erreurs d'encodage lèvent une *UnicodeError*. Les autres valeurs possibles sont 'ignore', 'replace' et tout autre nom enregistré via *codecs.register_error()*, voir la section *Gestionnaires d'erreurs*. Pour une liste des encodages possibles, voir la section *Standard Encodings*.

Note : Passer l'argument *encoding* à *str* permet de décoder tout *bytes-like object* directement, sans avoir besoin d'utiliser un *bytes* ou *bytearray* temporaire.

Modifié dans la version 3.1 : Gère les arguments nommés.

```
bytes.endswith(suffix[, start[, end]])
```

```
bytearray.endswith(suffix[, start[, end]])
```

Donne True si les octets se terminent par *suffix*, sinon False. *suffix* peut aussi être un tuple de suffixes à rechercher. Avec l'argument optionnel *start*, la recherche se fait à partir de cette position. Avec l'argument optionnel *end*, la comparaison s'arrête à cette position.

Les suffixes à rechercher peuvent être n'importe quel *bytes-like object*.

```
bytes.find(sub[, start[, end]])
```

```
bytearray.find(sub[, start[, end]])
```

Donne la première position où le *sub* se trouve dans les données, de telle sorte que *sub* soit contenue dans *s[start:end]*. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des *slices*. Donne -1 si *sub* n'est pas trouvé.

La sous-séquence à rechercher peut être un quelconque *bytes-like object* ou un nombre entier compris entre 0 et 255.

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

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

Modifié dans la version 3.3 : Accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

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

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

Comme *find()*, mais lève une *ValueError* lorsque la séquence est introuvable.

La sous-séquence à rechercher peut être un quelconque *bytes-like object* ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3 : Accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

```
bytes.join(iterable)
```

```
bytearray.join(iterable)
```

Donne un *bytes* ou *bytearray* qui est la concaténation des séquences de données binaires dans *iterable*. Une exception *TypeError* est levée si une valeur d'*iterable* n'est pas un *bytes-like objects*, y compris pour des *str*. Le séparateur entre les éléments est le contenu du *bytes* ou du *bytearray* depuis lequel cette méthode est appelée.

```
static bytes.maketrans(from, to)
```

```
static bytearray.maketrans(from, to)
```

Cette méthode statique renvoie une table de traduction utilisable par *bytes.translate()* qui permettra de changer chaque caractère de *from* par un caractère à la même position dans *to*; *from* et *to* doivent tous deux être des *bytes-like objects* et avoir la même longueur.

Nouveau dans la version 3.1.

`bytes.partition (sep)`

`bytearray.partition (sep)`

Divise la séquence à la première occurrence de *sep*, et renvoie un 3-tuple contenant la partie précédant le séparateur, le séparateur lui même (ou sa copie en *bytearray*), et la partie suivant le séparateur. Si le séparateur est pas trouvé, le 3-tuple renvoyé contiendra une copie de la séquence d'origine, suivi de deux *bytes* ou *bytearray* vides.

Le séparateur à rechercher peut être tout *bytes-like object*.

`bytes.replace (old, new[, count])`

`bytearray.replace (old, new[, count])`

Renvoie une copie de la séquence dont toutes les occurrences de la sous-séquence *old* sont remplacées par *new*. Si l'argument optionnel *count* est donné, seules les *count* premières occurrences de sont remplacés.

La sous-séquence à rechercher et son remplacement peuvent être n'importe quel *bytes-like object*.

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.rfind (sub[, start[, end]])`

`bytearray.rfind (sub[, start[, end]])`

Donne la plus grande position de *sub* dans la séquence, de telle sorte que *sub* soit dans *s[start:end]*. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des *slices*. Donne *-1* si *sub* n'est pas trouvable.

La sous-séquence à rechercher peut être un quelconque *bytes-like object* ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3 : Accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.rindex (sub[, start[, end]])`

`bytearray.rindex (sub[, start[, end]])`

Semblable à *rfind()* mais lève une *ValueError* lorsque *sub* est introuvable.

La sous-séquence à rechercher peut être un quelconque *bytes-like object* ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3 : Accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.rpartition (sep)`

`bytearray.rpartition (sep)`

Coupe la séquence à la dernière occurrence de *sep*, et renvoie un tuple de trois éléments contenant la partie précédent le séparateur, le séparateur lui même (ou sa copie, un *bytearray*), et la partie suivant le séparateur. Si le séparateur n'est pas trouvé, le tuple contiendra une copie de la séquence d'origine, suivi de deux *bytes* ou *bytesarray* vides.

Le séparateur à rechercher peut être tout *bytes-like object*.

`bytes.startswith (prefix[, start[, end]])`

`bytearray.startswith (prefix[, start[, end]])`

Donne *True* si les données binaires commencent par le *prefix* spécifié, sinon *False*. *prefix* peut aussi être un tuple de préfixes à rechercher. Avec l'argument *start* la recherche commence à cette position. Avec l'argument *end* option, la recherche s'arrête à cette position.

Le préfixe(s) à rechercher peuvent être n'importe quel *bytes-like object*.

`bytes.translate (table, delete=b'')`

`bytearray.translate (table, delete=b'')`

Renvoie une copie du *bytes* ou *bytearray* dont tous les octets de *delete* sont supprimés, et les octets restants changés par la table de correspondance donnée, qui doit être un objet *bytes* d'une longueur de 256.

Vous pouvez utiliser la méthode *bytes.maketrans()* pour créer une table de correspondance.

Donnez *None* comme *table* pour seulement supprimer des caractères :

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

Modifié dans la version 3.6 : *delete* est maintenant accepté comme argument nommé.

Les méthodes suivantes sur les *bytes* et *bytearray* supposent par défaut que les données traitées sont compatibles ASCII, mais peuvent toujours être utilisées avec des données binaires, arbitraires, en passant des arguments appropriés. Notez que toutes les méthodes de *bytearray* de cette section ne travaillent jamais sur l'objet lui-même, mais renvoient un nouvel objet.

`bytes.center(width[, fillbyte])`

`bytearray.center(width[, fillbyte])`

Renvoie une copie de l'objet centrée dans une séquence de longueur *width*. Le remplissage est fait en utilisant *fillbyte* (qui par défaut est un espace ASCII). Pour les objets *bytes*, la séquence initiale est renvoyée si *width* est inférieur ou égal à `len(s)`.

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.ljust(width[, fillbyte])`

`bytearray.ljust(width[, fillbyte])`

Renvoie une copie de l'objet aligné à gauche dans une séquence de longueur *width*. Le remplissage est fait en utilisant *fillbyte* (par défaut un espace ASCII). Pour les objets *bytes*, la séquence initiale est renvoyée si *width* est inférieure ou égale à `len(s)`.

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.lstrip([chars])`

`bytearray.lstrip([chars])`

Renvoie une copie de la séquence dont certains préfixes ont été supprimés. L'argument *chars* est une séquence binaire spécifiant le jeu d'octets à supprimer. Ce nom se réfère au fait de cette méthode est généralement utilisée avec des caractères ASCII. En cas d'omission ou *None*, la valeur par défaut de *chars* permet de supprimer des espaces ASCII. L'argument *chars* n'est pas un préfixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

La séquence de valeurs à supprimer peut être tout *bytes-like object*.

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.rjust(width[, fillbyte])`

`bytearray.rjust(width[, fillbyte])`

Renvoie une copie de l'objet justifié à droite dans une séquence de longueur *width*. Le remplissage est fait en utilisant le caractère *fillbyte* (par défaut est un espace ASCII). Pour les objets *bytes*, la séquence d'origine est renvoyée si *width* est inférieure ou égale à `len(s)`.

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.rsplit (sep=None, maxsplit=-1)`

`bytearray.rsplit (sep=None, maxsplit=-1)`

Divise la séquence d'octets en sous-séquences du même type, en utilisant *sep* comme séparateur. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être faites, celles « à droite ». Si *sep* est pas spécifié ou est `None`, toute sous-séquence composée uniquement d'espaces ASCII est un séparateur. En dehors du fait qu'il découpe par la droite, *rsplit()* se comporte comme *split()* qui est décrit en détail ci-dessous.

`bytes.rstrip ([chars])`

`bytearray.rstrip ([chars])`

Renvoie une copie de la séquence dont des octets finaux sont supprimés. L'argument *chars* est une séquence d'octets spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, les espaces ASCII sont supprimés. L'argument *chars* n'est pas un suffixe : toutes les combinaisons de ses valeurs sont retirées :

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

La séquence de valeurs à supprimer peut être tout *bytes-like object*.

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.split (sep=None, maxsplit=-1)`

`bytearray.split (sep=None, maxsplit=-1)`

Divise la séquence en sous-séquences du même type, en utilisant *sep* comme séparateur. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être faites (la liste aura donc au plus *maxsplit*+1 éléments), Si *maxsplit* n'est pas spécifié ou vaut -1, il n'y a aucune limite au nombre de découpes (elles sont toutes effectuées).

Si *sep* est donné, les délimiteurs consécutifs ne sont pas regroupés et ainsi délimitent ainsi des chaînes vides (par exemple, `b'1,,2'.split(b',')` donne `[b'1', b'', b'2']`). L'argument *sep* peut contenir plusieurs sous séquences (par exemple, `b'1<>2<>3'.split(b'<>')` renvoie `[b'1', b'2', b'3']`). Découper une chaîne vide en spécifiant *sep* donne `[b'']` ou `[bytearray(b'')]` en fonction du type de l'objet découpé. L'argument *sep* peut être n'importe quel *bytes-like object*.

Par exemple :

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,.'.split(b',')
[b'1', b'2', b'', b'3', b'']
```

Si *sep* n'est pas spécifié ou est `None`, un autre algorithme de découpe est appliqué : les espaces ASCII consécutifs sont considérés comme un seul séparateur, et le résultat ne contiendra pas les chaînes vides de début ou de la fin si la chaîne est préfixée ou suffixée d'espaces. Par conséquent, diviser une séquence vide ou une séquence composée d'espaces ASCII avec un séparateur `None` renvoie `[]`.

Par exemple :

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

`bytes.strip ([chars])`

`bytearray.strip([chars])`

Renvoie une copie de la séquence dont des caractères initiaux et finaux sont supprimés. L'argument *chars* est une séquence spécifiant le jeu d'octets à supprimer, le nom se réfère au fait de cette méthode est généralement utilisée avec des caractères ASCII. En cas d'omission ou `None`, les espaces ASCII sont supprimés. L'argument *chars* n'est ni un préfixe ni un suffixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> b'   spacious   '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

La séquence de valeurs à supprimer peut être tout *bytes-like object*.

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

Les méthodes suivantes sur les *bytes* et *bytearray* supposent l'utilisation d'un format binaire compatible ASCII, et donc doivent être évités lorsque vous travaillez avec des données binaires arbitraires. Notez que toutes les méthodes de *bytearray* de cette section *ne modifient pas* les octets, ils produisent de nouveaux objets.

`bytes.capitalize()`

`bytearray.capitalize()`

Renvoie une copie de la séquence dont chaque octet est interprété comme un caractère ASCII, le premier octet en capitale et le reste en minuscules. Les octets non ASCII ne sont pas modifiés.

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.expandtabs(tabsize=8)`

`bytearray.expandtabs(tabsize=8)`

Renvoie une copie de la séquence où toutes les tabulations ASCII sont remplacées par un ou plusieurs espaces ASCII, en fonction de la colonne courante et de la taille de tabulation donnée. Les positions des tabulations se trouvent tous les *tabsize* caractères (8 par défaut, ce qui donne les positions de tabulations aux colonnes 0, 8, 16 et ainsi de suite). Pour travailler sur la séquence, la colonne en cours est mise à zéro et la séquence est examinée octets par octets. Si l'octet est une tabulation ASCII (`b' '`), un ou plusieurs espaces sont insérés au résultat jusqu'à ce que la colonne courante soit égale à la position de tabulation suivante. (Le caractère tabulation lui-même n'est pas copié.) Si l'octet courant est un saut de ligne ASCII (`b' '`) ou un retour chariot (`b'\r'`), il est copié et la colonne en cours est remise à zéro. Tout autre octet est copié inchangé et la colonne en cours est incrémentée de un indépendamment de la façon dont l'octet est représenté lors de l'affichage :

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

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.isalnum()`

`bytearray.isalnum()`

Donne `True` si tous les caractères de la chaîne sont des caractères ASCII alphabétiques ou chiffres. et que la séquence n'est pas vide, sinon `False`. Les caractères ASCII alphabétiques sont les suivants `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` et les chiffres : `b'0123456789'`.

Par exemple :

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

`bytes.isalpha()`

`bytearray.isalpha()`

Donne True si tous les octets dans la séquence sont des caractères alphabétiques ASCII et la que la séquence n'est pas vide, sinon False. Les caractères ASCII alphabétiques sont : `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Par exemple :

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

`bytes.isdigit()`

`bytearray.isdigit()`

Donne True si tous les octets de la séquence sont des chiffres ASCII et que la séquence n'est pas vide, sinon False. Les chiffres ASCII sont `b'0123456789'`.

Par exemple :

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

`bytes.islower()`

`bytearray.islower()`

Donne True s'il y a au moins un caractère ASCII minuscule dans la séquence et aucune capitale, sinon False.

Par exemple :

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.isspace()`

`bytearray.isspace()`

Donne True si tous les octets de la séquence sont des espaces ASCII et que la séquence n'est pas vide, sinon False. Les espèces ASCII sont `b' \t\n\r\x0b\f'` (espace, tabulation, saut de ligne, retour chariot, tabulation verticale, *form feed*).

`bytes.istitle()`

`bytearray.istitle()`

Donne True si la séquence ASCII est *titlecased*, et qu'elle ne soit pas vide, sinon False. Voir `bytes.title()` pour plus de détails sur la définition de *titlecase*.

Par exemple :

```
>>> b'Hello World'.istitle()
True
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> b'Hello world'.istitle()
False
```

`bytes.isupper()``bytearray.isupper()`

Donne True s'il y a au moins un caractère alphabétique majuscule ASCII dans la séquence et aucun caractères ASCII minuscules, sinon False.

Par exemple :

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.lower()``bytearray.lower()`

Renvoie une copie de la séquence dont tous les caractères ASCII en majuscules sont convertis en leur équivalent en minuscules.

Par exemple :

```
>>> b'Hello World'.lower()
b'hello world'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note : La version `bytearray` de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.splitlines(keepends=False)``bytearray.splitlines(keepends=False)`

Renvoie une liste des lignes de la séquence d'octets, découpant au niveau des fin de lignes ASCII. Cette méthode utilise l'approche *universal newlines* pour découper les lignes. Les fins de ligne ne sont pas inclus dans la liste des résultats, sauf si `keepends` est donné et vrai.

Par exemple :

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

Contrairement à `split()` lorsque le délimiteur `sep` est fourni, cette méthode renvoie une liste vide pour la chaîne vide, et un saut de ligne à la fin ne se traduit pas par une ligne supplémentaire :

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

`bytes.swapcase()``bytearray.swapcase()`

Renvoie une copie de la séquence dont tous les caractères ASCII minuscules sont convertis en majuscules et vice-versa.

Par exemple :

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Contrairement à `str.swapcase()`, `bin.swapcase().swapcase() == bin` est toujours vrai. Les conversions majuscule/minuscule en ASCII étant toujours symétrique, ce qui n'est pas toujours vrai avec Unicode.

Note : La version `bytearray` de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.title()`

`bytearray.title()`

Renvoie une version *titlecased* de la séquence d'octets où les mots commencent par un caractère ASCII majuscule et les caractères restants sont en minuscules. Les octets non capitalisables ne sont pas modifiés.

Par exemple :

```
>>> b'Hello world'.title()
b'Hello World'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les caractères ASCII majuscules sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. Aucun autre octet n'est capitalisable.

Pour l'algorithme, la notion de mot est définie simplement et indépendamment de la langue comme un groupe de lettres consécutives. La définition fonctionne dans de nombreux contextes, mais cela signifie que les apostrophes (typiquement de la forme possessive en Anglais) forment les limites de mot, ce qui n'est pas toujours le résultat souhaité :

```
>>> b"they're bill's friends from the UK".title()
b'They'Re Bill'S Friends From The Uk'
```

Une solution pour contourner le problème des apostrophes peut être obtenue en utilisant des expressions rationnelles :

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                     lambda mo: mo.group(0)[0:1].upper() +
...                                 mo.group(0)[1:].lower(),
...                     s)
...
>>> titlecase(b"they're bill's friends.")
b'They'Re Bill'S Friends.'
```

Note : La version `bytearray` de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.upper()`

`bytearray.upper()`

Renvoie une copie de la séquence dont tous les caractères ASCII minuscules sont convertis en leur équivalent majuscule.

Par exemple :

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.zfill (width)`

`bytearray.zfill (width)`

Return a copy of the sequence left filled with ASCII `b'0'` digits to make a sequence of length *width*. A leading sign prefix (`b'+' / b'-'`) is handled by inserting the padding *after* the sign character rather than before. For *bytes* objects, the original sequence is returned if *width* is less than or equal to `len(seq)`.

Par exemple :

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

Note : La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

4.8.4 Formatage de *bytes* a la `printf`

Note : Les opérations de formatage décrites ici présentent une variété de bizarreries qui conduisent à un certain nombre d'erreurs classiques (typiquement, échouer à afficher des tuples ou des dictionnaires correctement). Si la valeur à afficher peut être un tuple ou un dictionnaire, mettez le a l'intérieur d'un autre tuple.

Les objets *bytes* (`bytes` et `bytearray`) ont un unique opérateur : l'opérateur `%` (modulo). Il est aussi connu sous le nom d'opérateur de mise en forme. Avec `format % values` (où *format* est un objet *bytes*), les marqueurs de conversion `%` dans *format* sont remplacés par zéro ou plus de *values*. L'effet est similaire à la fonction `sprintf()` du langage C.

Si *format* ne nécessite qu'un seul argument, *values* peut être un objet unique.⁵ Si *values* est un tuple, il doit contenir exactement le nombre d'éléments spécifiés dans le format en *bytes*, ou un seul objet de correspondances (*mapping object*, par exemple, un dictionnaire).

Un indicateur de conversion contient deux ou plusieurs caractères et comporte les éléments suivants, qui doivent apparaître dans cet ordre :

1. Le caractère `'%'`, qui marque le début du marqueur.
2. La clé de correspondance (facultative), composée d'une suite de caractères entre parenthèse (par exemple, `(somename)`).
3. Des options de conversion, facultatives, qui affectent le résultat de certains types de conversion.
4. Largeur minimum (facultative). Si elle vaut `'*'` (astérisque), la largeur est lue de l'élément suivant du tuple *values*, et l'objet à convertir vient après la largeur de champ minimale et la précision facultative.
5. Précision (facultatif), donnée sous la forme d'un `'.'` (point) suivi de la précision. Si la précision est `'*'` (un astérisque), la précision est lue à partir de l'élément suivant du tuple *values* et la valeur à convertir vient ensuite.
6. Modificateur de longueur (facultatif).

7. Type de conversion.

Lorsque l'argument de droite est un dictionnaire (ou un autre type de *mapping*), les marqueurs dans le *bytes* doivent inclure une clé présente dans le dictionnaire, écrite entre parenthèses, immédiatement après le caractère '%'. La clé indique quelle valeur du dictionnaire doit être formatée. Par exemple :

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b'Python', b'number': 2})
b'Python has 002 quote types.'
```

Dans ce cas, aucune * ne peuvent se trouver dans le format (car ces * nécessitent une liste (accès séquentiel) de paramètres).

Les caractères indicateurs de conversion sont :

Option	Signification
'#'	La conversion utilisera la « forme alternative » (définie ci-dessous).
'0'	Les valeurs numériques converties seront complétées de zéros.
'-'	La valeur convertie est ajustée à gauche (remplace la conversion '0' si les deux sont données).
' '	(un espace) Un espace doit être laissé avant un nombre positif (ou chaîne vide) produite par la conversion d'une valeur signée.
'+'	Un caractère de signe ('+' ou '-') précède la valeur convertie (remplace le marqueur « espace »).

Un modificateur de longueur (h, l ou L) peut être présent, mais est ignoré car il est pas nécessaire pour Python, donc par exemple %ld est identique à %d.

Les types utilisables dans les conversion sont :

Conversion	Signification	Notes
'd'	Entier décimal signé.	
'i'	Entier décimal signé.	
'o'	Valeur octale signée.	(1)
'u'	Type obsolète — identique à 'd'.	(8)
'x'	Hexadécimal signé (en minuscules).	(2)
'X'	Hexadécimal signé (en capitales).	(2)
'e'	Format exponentiel pour un <i>float</i> (minuscule).	(3)
'E'	Format exponentiel pour un <i>float</i> (en capitales).	(3)
'f'	Format décimal pour un <i>float</i> .	(3)
'F'	Format décimal pour un <i>float</i> .	(3)
'g'	Format <i>float</i> . Utilise le format exponentiel minuscules si l'exposant est inférieur à -4 ou pas plus petit que la précision, sinon le format décimal.	(4)
'G'	Format <i>float</i> . Utilise le format exponentiel en capitales si l'exposant est inférieur à -4 ou pas plus petit que la précision, sinon le format décimal.	(4)
'c'	Octet simple (Accepte un nombre entier ou un seul objet <i>byte</i>).	
'b'	<i>Bytes</i> (tout objet respectant le buffer protocol ou ayant la méthode <code>__bytes__()</code>).	(5)
's'	's' est un alias de 'b' et ne devrait être utilisé que pour du code Python2/3.	(6)
'a'	<i>Bytes</i> (convertit n'importe quel objet Python en utilisant <code>repr(obj).encode('ascii', 'backslashreplace')</code>).	(5)
'r'	'r' est un alias de 'a' et ne devrait être utilisé que dans du code Python2/3.	(7)
'%'	Aucun argument n'est converti, donne un caractère de '%' dans le résultat.	

Notes :

- (1) La forme alternative entraîne l'insertion d'un préfixe octal ('0o') avant le premier chiffre.
- (2) La forme alternative entraîne l'insertion d'un préfixe '0x' ou '0X' (respectivement pour les formats 'x' et 'X') avant le premier chiffre.
- (3) La forme alternative implique la présence d'un point décimal, même si aucun chiffre ne le suit.
La précision détermine le nombre de chiffres après la virgule, 6 par défaut.
- (4) La forme alternative implique la présence d'un point décimal et les zéros non significatifs sont conservés (ils ne le seraient pas autrement).
La précision détermine le nombre de chiffres significatifs avant et après la virgule. 6 par défaut.
- (5) Si la précision est N, la sortie est tronquée à N caractères.
- (6) `b'%s'` est obsolète, mais ne sera pas retiré des version 3.x.
- (7) `b'%r'` est obsolète mais ne sera pas retiré dans Python 3.x.
- (8) Voir la [PEP 237](#).

Note : La version `bytearray` de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

Voir aussi :

[PEP 461](#) – Ajout du formatage via % aux `bytes` et `bytearray`

Nouveau dans la version 3.5.

4.8.5 Vues de mémoires

Les `memoryview` permettent à du code Python d'accéder sans copie aux données internes d'un objet prenant en charge le `buffer protocol`.

class `memoryview` (*obj*)

Crée une `memoryview` faisant référence à *obj*. *obj* doit supporter le `buffer protocol`. Les objets natifs prenant en charge le `buffer protocol` sont `bytes` et `bytearray`.

Une `memoryview` a la notion d'*élément*, qui est l'unité de mémoire atomique gérée par l'objet *obj* d'origine. Pour de nombreux types simples comme `bytes` et `bytearray`, l'élément est l'octet, mais pour d'autres types tels que `array.array` les éléments peuvent être plus grands.

`len(view)` est égal à la grandeur de `tolist`. Si `view.ndim = 0`, la longueur vaut 1. Si `view.ndim = 1`, la longueur est égale au nombre d'éléments de la vue. Pour les dimensions plus grandes, la longueur est égale à la longueur de la sous-liste représentée par la vue. L'attribut `itemsize` vous donnera la taille en octets d'un élément.

Une `memoryview` autorise le découpage et l'indexage de ses données. Découper sur une dimension donnera une sous-vue :

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

Si le `format` est un des formats natif du module `struct`, indexer avec un nombre entier ou un *tuple* de nombres entiers est aussi autorisé et renvoie un seul *élément* du bon type. Les `memoryview` à une dimension peuvent être

indexées avec un nombre entier ou un *tuple* d'un entier. Les *memoryview* multi-dimensionnelles peuvent être indexées avec des *tuples* d'exactly *ndim* entiers où *ndim* est le nombre de dimensions. Les *memoryviews* à zéro dimension peuvent être indexées avec un *tuple* vide.

Voici un exemple avec un autre format que *byte* :

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

Si l'objet sous-jacent est accessible en écriture, la *memoryview* autorisera les assignations de tranches à une dimension. Redimensionner n'est cependant pas autorisé :

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

Les *memoryviews* à une dimension de types hachables (lecture seule) avec les formats “B”, “b”, ou “c” sont aussi hachables. La fonction de hachage est définie tel que `hash(m) == hash(m.tobytes())` :

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::2]) == hash(b'abcefg'[::2])
True
```

Modifié dans la version 3.3 : Les *memoryviews* à une dimension peuvent aussi être découpées. Les *memoryviews* à une dimension avec les formats “B”, “b”, ou “c” sont maintenant hachables.

Modifié dans la version 3.4 : *memoryview* est maintenant enregistrée automatiquement avec `collections.abc.Sequence`

Modifié dans la version 3.5 : les *memoryviews* peuvent maintenant être indexées par un n-uplet d'entiers.

La *memoryview* dispose de plusieurs méthodes :

`__eq__` (exporter)

Une *memoryview* et un *exporter* de la **PEP 3118** sont égaux si leurs formes sont équivalentes et si toutes les valeurs correspondantes sont égales, le format respectifs des opérandes étant interprétés en utilisant la syntaxe de *struct*.

Pour le sous-ensemble des formats de `struct` supportés par `tolist()`, `v` et `w` sont égaux si `v.tolist() == w.tolist()` :

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

Si l'un des format n'est pas supporté par le module de `struct`, les objets seront toujours considérés différents (même si les formats et les valeurs contenues sont identiques) :

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Notez que pour les `memoryview`, comme pour les nombres à virgule flottante, `v is w` n'implique pas `v == w`.

Modifié dans la version 3.3 : Les versions précédentes comparaient la mémoire brute sans tenir compte du format de l'objet ni de sa structure logique.

tobytes()

Renvoie les données du *buffer* sous forme de *bytes*. Cela équivaut à appeler le constructeur `bytes` sur le `memoryview`.

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

Pour les listes non contiguës le résultat est égal à la représentation en liste aplatie dont tous les éléments sont convertis en octets. `tobytes()` supporte toutes les chaînes de format, y compris celles qui ne sont pas connues du module `struct`.

hex()

Renvoie une chaîne contenant deux chiffres hexadécimaux pour chaque octet de la mémoire.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

Nouveau dans la version 3.5.

tolist()

Renvoie les données de la mémoire sous la forme d'une liste d'éléments.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Modifié dans la version 3.3 : `tolist()` prend désormais en charge tous les formats d'un caractère du module `struct` ainsi que des représentations multidimensionnelles.

release()

Libère le tampon sous-jacent exposé par l'objet `memoryview`. Beaucoup d'objets prennent des initiatives particulières lorsqu'ils sont liés à une vue (par exemple, un `bytearray` refusera temporairement de se faire redimensionner). Par conséquent, appeler `release()` peut être pratique pour lever ces restrictions (et en libérer les ressources liées) aussi tôt que possible.

Après le premier appel de cette méthode, toute nouvelle opération sur la `view` lève une `ValueError` (sauf `release()` elle-même qui peut être appelée plusieurs fois) :

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Le protocole de gestion de contexte peut être utilisé pour obtenir un effet similaire, via l'instruction `with` :

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Nouveau dans la version 3.2.

cast(format[, shape])

Change le format ou la forme d'une `memoryview`. Par défaut `shape` vaut `[byte_length//new_itemsize]`, ce qui signifie que la vue résultante n'aura qu'une dimension. La valeur renvoyée est une nouvelle `memoryview`, mais la mémoire elle-même n'est pas copiée. Les changements supportés sont une dimension vers *C-contiguous* et *C-contiguous* vers une dimension.

Le format de destination est limité à un seul élément natif de la syntaxe du module `struct`. L'un des formats doit être un *byte* ("B", "b", ou "c"). La longueur du résultat en octets doit être la même que la longueur initiale.

Transforme *1D/long* en *1D/unsigned bytes* :

```
>>> import array
>>> a = array.array('l', [1, 2, 3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
```

(suite sur la page suivante)

(suite de la page précédente)

```

3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24

```

Transforme *1D/unsigned bytes* en *1D/char* :

```

>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')

```

Transforme *1D/bytes* en *3D/ints* en *1D/signed char* :

```

>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48

```

Transforme *1D/unsigned char* en *2D/unsigned long* :

```

>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)

```

(suite sur la page suivante)

(suite de la page précédente)

```

2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]

```

Nouveau dans la version 3.3.

Modifié dans la version 3.5 : Le format de la source n'est plus restreint lors de la transformation vers une vue d'octets.

Plusieurs attributs en lecture seule sont également disponibles :

obj

L'objet sous-jacent de la *memoryview* :

```

>>> b = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True

```

Nouveau dans la version 3.3.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. This is the amount of space in bytes that the array would use in a contiguous representation. It is not necessarily equal to `len(m)` :

```

>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12

```

Tableaux multidimensionnels :

```

>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96

```

Nouveau dans la version 3.3.

readonly

Un booléen indiquant si la mémoire est en lecture seule.

format

Une chaîne contenant le format (dans le style de *struct*) pour chaque élément de la vue. Une *memoryview*

peut être créée depuis des exportateurs de formats arbitraires, mais certaines méthodes (comme `tolist()`) sont limitées aux formats natifs à un seul élément.

Modifié dans la version 3.3 : le format 'B' est maintenant traité selon la syntaxe du module `struct`. Cela signifie que `memoryview(b'abc')[0] == b'abc'[0] == 97`.

itemsize

La taille en octets de chaque élément d'une *memoryview* :

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

Un nombre entier indiquant le nombre de dimensions d'un tableau multi-dimensionnel représenté par la *memoryview*.

shape

Un *tuple* d'entiers de longueur *ndim* donnant la forme de la *memoryview* sous forme d'un tableau à N dimensions.

Modifié dans la version 3.3 : Un *tuple* vide au lieu de `None` lorsque *ndim* = 0.

strides

Un *tuple* d'entiers de longueur *ndim* donnant la taille en octets permettant d'accéder à chaque dimensions du tableau.

Modifié dans la version 3.3 : Un *tuple* vide au lieu de `None` lorsque *ndim* = 0.

suboffsets

Détail de l'implémentation des *PIL-style arrays*. La valeur n'est donné qu'à titre d'information.

c_contiguous

Un booléen indiquant si la mémoire est *C-contiguous*.

Nouveau dans la version 3.3.

f_contiguous

Un booléen indiquant si la mémoire est Fortran *contiguous*.

Nouveau dans la version 3.3.

contiguous

Un booléen indiquant si la mémoire est *contiguous*.

Nouveau dans la version 3.3.

4.9 Types d'ensembles — set, frozenset

Un objet *set* est une collection non-triée d'objets *hashable* distincts. Les utilisations classiques sont le test d'appartenance, la déduplication d'une séquence, ou le calcul d'opérations mathématiques telles que l'intersection, l'union, la différence, ou la différence symétrique. (Pour les autres conteneurs, voir les classes natives *dict*, *list*, et *tuple*, ainsi que le module *collections*.)

Comme pour les autres collections, les ensembles supportent `x in set`, `len(set)`, et `for x in set`. En tant que collection non-triée, les ensembles n'enregistrent pas la position des éléments ou leur ordre d'insertion. En conséquence, les *sets* n'autorisent ni l'indexation, ni le découpage, ou tout autre comportement de séquence.

Il existe actuellement deux types natifs pour les ensembles, *set* et *frozenset*. Le type *set* est muable — son contenu peut changer en utilisant des méthodes comme `add()` et `remove()`. Puisqu'il est muable, il n'a pas de valeur de hachage et ne peut donc pas être utilisé ni comme clef de dictionnaire ni comme élément d'un autre ensemble. Le type

frozenset est immuable et *hashable* — son contenu ne peut être modifié après sa création, il peut ainsi être utilisé comme clef de dictionnaire ou élément d'un autre *set*.

Des *sets* (mais pas des *frozensets*) peuvent être créés par une liste d'éléments séparés par des virgules et entre accolades, par exemple : `{ 'jack', 'sjoerd' }`, en plus du constructeur de la classe *set*.

Les constructeurs des deux classes fonctionnent pareil :

```
class set ([iterable])
class frozenset ([iterable])
```

Renvoie un nouveau *set* ou *frozenset* dont les éléments viennent d'*iterable*. Les éléments d'un *set* doivent être *hashable*. Pour représenter des *sets* de *sets* les *sets* intérieurs doivent être des *frozenset*. Si *iterable* n'est pas spécifié, un nouveau *set* vide est renvoyé.

Les instances de *set* et *frozenset* fournissent les opérations suivantes :

```
len(s)
```

Donne le nombre d'éléments dans le *set* *s* (cardinalité de *s*).

```
x in s
```

Test d'appartenance de *x* dans *s*.

```
x not in s
```

Test de non-appartenance de *x* dans *s*.

```
isdisjoint (other)
```

Renvoie `True` si l'ensemble n'a aucun élément en commun avec *other*. Les ensembles sont disjoints si et seulement si leur intersection est un ensemble vide.

```
issubset (other)
```

```
set <= other
```

Teste si tous les éléments du *set* sont dans *other*.

```
set < other
```

Teste si l'ensemble est un sous-ensemble de *other*, c'est-à-dire, `set <= other and set != other`.

```
issuperset (other)
```

```
set >= other
```

Teste si tous les éléments de *other* sont dans l'ensemble.

```
set > other
```

Teste si l'ensemble est un sur-ensemble de *other*, c'est-à-dire, `set >= other and set != other`.

```
union (*others)
```

```
set | other | ...
```

Renvoie un nouvel ensemble dont les éléments viennent de l'ensemble et de tous les autres.

```
intersection (*others)
```

```
set & other & ...
```

Renvoie un nouvel ensemble dont les éléments sont communs à l'ensemble et à tous les autres.

```
difference (*others)
```

```
set - other - ...
```

Renvoie un nouvel ensemble dont les éléments sont dans l'ensemble mais ne sont dans aucun des autres.

```
symmetric_difference (other)
```

```
set ^ other
```

Renvoie un nouvel ensemble dont les éléments sont soit dans l'ensemble, soit dans les autres, mais pas dans les deux.

```
copy ()
```

Renvoie un nouvel ensemble, copie de surface de *s*.

Remarque : Les méthodes `union()`, `intersection()`, `difference()`, et `symmetric_difference()`, `issubset()`, et `issuperset()` acceptent n'importe quel itérable comme argument, contrairement aux opérateurs équivalents qui n'acceptent que des *sets*. Il est donc préférable d'éviter les constructions comme `set('abc') & 'cbs'`, sources typiques d'erreurs, en faveur d'une construction plus lisible : `set('abc').intersection('cbs')`.

Les classes `set` et `frozenset` supportent les comparaisons d'ensemble à ensemble. Deux ensembles sont égaux si et seulement si chaque éléments de chaque ensemble est contenu dans l'autre (autrement dit que chaque ensemble est un sous-ensemble de l'autre). Un ensemble est plus petit qu'un autre ensemble si et seulement si le premier est un sous-ensemble du second (un sous-ensemble, mais pas égal). Un ensemble est plus grand qu'un autre ensemble si et seulement si le premier est un sur-ensemble du second (est un sur-ensemble mais n'est pas égal).

Les instances de `set` se comparent aux instances de `frozenset` en fonction de leurs membres. Par exemple, `set('abc') == frozenset('abc')` envoie `True`, ainsi que `set('abc') in set([frozenset('abc')])`.

Les comparaisons de sous-ensemble et d'égalité ne se généralisent pas en une fonction donnant un ordre total. Par exemple, deux ensemble disjoints non vides ne sont ni égaux et ni des sous-ensembles l'un de l'autre, donc toutes ces comparaisons donnent `False` : `a < b`, `a == b`, et `a > b`.

Puisque les `sets` ne définissent qu'un ordre partiel (par leurs relations de sous-ensembles), la sortie de la méthode `list.sort()` n'est pas définie pour des listes d'ensembles.

Les éléments des `sets`, comme les clefs de dictionnaires, doivent être *hashable*.

Les opérations binaires mélangeant des instances de `set` et `frozenset` renvoient le type de la première opérande. Par exemple : `frozenset('ab') | set('bc')` renvoie une instance de `frozenset`.

La table suivante liste les opérations disponibles pour les `set` mais qui ne s'appliquent pas aux instances de `frozenset` :

update (*others)

set |= other | ...

Met à jour l'ensemble, ajoutant les éléments de tous les autres.

intersection_update (*others)

set &= other & ...

Met à jour l'ensemble, ne gardant que les éléments trouvés dans tous les autres.

difference_update (*others)

set -= other | ...

Met à jour l'ensemble, retirant les éléments trouvés dans les autres.

symmetric_difference_update (other)

set ^= other

Met à jour le set, ne gardant que les éléments trouvés dans un des ensembles mais pas dans les deux.

add (elem)

Ajoute l'élément *elem* au set.

remove (elem)

Retire l'élément *elem* de l'ensemble. Lève une exception `KeyError` si *elem* n'est pas dans l'ensemble.

discard (elem)

Retire l'élément *elem* de l'ensemble s'il y est.

pop ()

Retire et renvoie un élément arbitraire de l'ensemble. Lève une exception `KeyError` si l'ensemble est vide.

clear ()

Supprime tous les éléments du *set*.

Notez que les versions non-opérateurs des méthodes `update()`, `intersection_update()`, `difference_update()`, et `symmetric_difference_update()` acceptent n'importe quel itérable comme argument.

Notez que l'argument *elem* des méthodes `__contains__()`, `remove()`, et `discard()` peut être un ensemble. Pour supporter la recherche d'un `frozenset` équivalent, un `frozenset` temporaire est créé depuis *elem*.

4.10 Les types de correspondances — dict

Un objet *mapping* fait correspondre des valeurs *hashable* à des objets arbitraires. Les *mappings* sont des objets muables. Il n'existe pour le moment qu'un type de *mapping* standard, le *dictionary*. (Pour les autres conteneurs, voir les types natifs *list*, *set*, et *tuple*, ainsi que le module *collections*.)

Les clefs d'un dictionnaire sont *presque* des données arbitraires. Les valeurs qui ne sont pas *hashable*, c'est-à-dire qui contiennent les listes, des dictionnaires ou autre type muable (qui sont comparés par valeur plutôt que par leur identité) ne peuvent pas être utilisées comme clef de dictionnaire. Les types numériques utilisés comme clef obéissent aux règles classiques en ce qui concerne les comparaisons : si deux nombres sont égaux (comme 1 et 1.0) ils peuvent tous les deux être utilisés pour obtenir la même entrée d'un dictionnaire. (Notez cependant que puisque les ordinateurs stockent les nombres à virgule flottante sous forme d'approximations, il est généralement imprudent de les utiliser comme clefs de dictionnaires.)

Il est possible de créer des dictionnaires en plaçant entre accolades une liste de paires de *key* : *value* séparés par des virgules, par exemple : `{'jack': 4098, 'sjoerd': 4127}` ou `{4098: 'jack', 4127: 'sjoerd'}`, ou en utilisant le constructeur de *dict*.

```
class dict (**kwarg)
```

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

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

Renvoie un nouveau dictionnaire initialisé depuis un argument positionnel optionnel, et un ensemble (vide ou non) d'arguments par mot clef.

Si aucun argument positionnel n'est donné, un dictionnaire vide est créé. Si un argument positionnel est donné et est un *mapping object*, un dictionnaire est créé avec les mêmes paires de clef-valeurs que le *mapping* donné. Autrement, l'argument positionnel doit être un objet *iterable*. Chaque élément de cet itérable doit lui-même être un itérable contenant exactement deux objets. Le premier objet de chaque élément devient la une clef du nouveau dictionnaire, et le second devient sa valeur correspondante. Si une clef apparaît plus d'une fois, la dernière valeur pour cette clef devient la valeur correspondante à cette clef dans le nouveau dictionnaire.

Si des arguments nommés sont donnés, ils sont ajoutés au dictionnaire créé depuis l'argument positionnel. Si une clef est déjà présente, la valeur de l'argument nommé remplace la valeur reçue par l'argument positionnel.

Typiquement, les exemples suivants renvoient tous un dictionnaire valant `{"one": 1, "two": 2, "three": 3}` :

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Fournir les arguments nommés comme dans le premier exemple en fonctionne que pour des clefs qui sont des identifiants valide en Python. Dans les autres cas, toutes les clefs valides sont utilisables.

Voici les opérations gérées par les dictionnaires, (par conséquent, d'autres types de *mapping* peuvent les gérer aussi) :

len(d)

Renvoie le nombre d'éléments dans le dictionnaire *d*.

d[key]

Donne l'élément de *d* dont la clef est *key*. Lève une exception *KeyError* si *key* n'est pas dans le dictionnaire. Si une sous-classe de *dict* définit une méthode `__missing__()` et que *key* manque, l'opération `d[key]` appelle cette méthode avec la clef *key* en argument. L'opération `d[key]` renverra la valeur, ou lèvera l'exception renvoyée ou levée par l'appel à `__missing__(key)`. Aucune autre opération ni méthode n'appellent `__missing__()`. If `__missing__()` n'est pas définie, une exception *KeyError* est levée. `__missing__()` doit être une méthode; ça ne peut être une variable d'instance :

```

>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1

```

L'exemple ci-dessus montre une partie de l'implémentation de `collections.Counter`. `collections.defaultdict` implémente aussi `__missing__`.

d[key] = value

Assigne `d[key]` à `value`.

del d[key]

Supprime `d[key]` de `d`. Lève une exception `KeyError` si `key` n'est pas dans le dictionnaire.

key in d

Renvoie `True` si `d` a la clef `key`, sinon `False`.

key not in d

Équivalent à `not key in d`.

iter(d)

Renvoie un itérateur sur les clefs du dictionnaire. C'est un raccourci pour `iter(d.keys())`.

clear()

Supprime tous les éléments du dictionnaire.

copy()

Renvoie une copie de surface du dictionnaire.

classmethod fromkeys(seq[, value])

Crée un nouveau dictionnaire avec les clefs de `seq` et les valeurs à `value`.

`fromkeys()` est une *class method* qui renvoie un nouveau dictionnaire. `value` vaut `None` par défaut.

get(key[, default])

Renvoie la valeur de `key` si `key` est dans le dictionnaire, sinon `default`. Si `default` n'est pas donné, il vaut `None` par défaut, de manière à ce que cette méthode ne lève jamais `KeyError`.

items()

Renvoie une nouvelle vue des éléments du dictionnaire (paires de `(key, value)`). Voir la [documentation des vues](#).

keys()

Renvoie une nouvelle vue des clefs du dictionnaire. Voir la [documentation des vues](#).

pop(key[, default])

Si `key` est dans le dictionnaire elle est supprimée et sa valeur est renvoyée, sinon renvoie `default`. Si `default` n'est pas donné et que `key` n'est pas dans le dictionnaire, une `KeyError` est levée.

popitem()

Supprime et renvoie une `(key, value)` arbitraire du dictionnaire.

`popitem()` est pratique pour itérer un dictionnaire de manière destructive, comme souvent dans les algorithmes sur les ensembles. Si le dictionnaire est vide, appeler `popitem()` lève une `KeyError`.

setdefault(key[, default])

Si `key` est dans le dictionnaire, sa valeur est renvoyée. Sinon, insère `key` avec comme valeur `default` et renvoie `default`. `default` vaut `None` par défaut.

update([other])

Met à jour le dictionnaire avec les paires de clef/valeur d'`other`, écrasant les clefs existantes. Renvoie `None`.

`update()` accepte aussi bien un autre dictionnaire qu'un itérable de clef/valeurs (sous forme de *tuples* ou autre itérables de longueur deux). Si des paramètres par mot-clef sont donnés, le dictionnaire est ensuite mis à jour avec ces paires de clef/valeurs : `d.update(red=1, blue=2)`.

values()

Renvoie une nouvelle vue des valeurs du dictionnaire. Voir la [documentation des vues](#).

Deux dictionnaires sont égaux si et seulement si ils ont les mêmes paires de clef-valeur. Les comparaisons d'ordre (<, <=, >=, >) lèvent une `TypeError`.

Voir aussi :

`types.MappingProxyType` peut être utilisé pour créer une vue en lecture seule d'un `dict`.

4.10.1 Les vues de dictionnaires

Les objets renvoyés par `dict.keys()`, `dict.values()` et `dict.items()` sont des *vues*. Ils fournissent une vue dynamique des éléments du dictionnaire, ce qui signifie que si le dictionnaire change, la vue reflète ces changements.

Les vues de dictionnaires peuvent être itérées et ainsi renvoyer les données du dictionnaire, elle gèrent aussi les tests de présence :

len(dictview)

Renvoie le nombre d'entrées du dictionnaire.

iter(dictview)

Renvoie un itérateur sur les clefs, les valeurs, ou les éléments (représentés par des *tuples* de (key, value) du dictionnaire.

Les clefs et les valeurs sont itérées dans un ordre arbitraire qui n'est pas aléatoire, qui peut varier d'une implémentation de Python à l'autre, et qui dépend de l'historique d'insertion et de suppressions d'éléments. Si les vues de clefs, de valeurs, et d'éléments sont parcourues sans que le dictionnaire ne soit modifié, l'ordre des éléments correspondra directement. Ceci permet la création de paires de (key, value) en utilisant `zip()` : `pairs = zip(d.values(), d.keys())`. Un autre moyen de construire la même liste est `pairs = [(v, k) for (k, v) in d.items()]`.

Parcourir des vues tout en ajoutant ou supprimant des entrées dans un dictionnaire peut lever une `RuntimeError` ou ne pas fournir toutes les entrées.

x in dictview

Renvoie `True` si `x` est dans les clefs, les valeurs, ou les éléments du dictionnaire sous-jacent (dans le dernier cas, `x` doit être un *tuple* (key, value)).

Les vues de clefs sont semblables à des ensembles puisque leurs entrées sont uniques et hachables. Si toutes les valeurs sont hachables, et qu'ainsi toutes les paires de (key, value) sont uniques et hachables, alors la vue donnée par `items()` est aussi semblable à un ensemble. (Les vues données par `items()` ne sont généralement pas traitées comme des ensembles, car leurs valeurs ne sont généralement pas uniques.) Pour les vues semblables aux ensembles, toutes les opérations définies dans la classe de base abstraite `collections.abc.Set` sont disponibles (comme `==`, `<`, ou `^`).

Exemple d'utilisation de vue de dictionnaire :

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order
>>> list(keys)
```

(suite sur la page suivante)

(suite de la page précédente)

```

['eggs', 'bacon', 'sausage', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['spam', 'bacon']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}

```

4.11 Le type gestionnaire de contexte

L'instruction `with` permet l'existence de contextes définis à l'exécution par des gestionnaires de contextes. C'est implémenté via une paire de méthodes permettant de définir un contexte, à l'exécution, qui est entré avant l'exécution du corps de l'instruction, et qui est quitté lorsque l'instruction se termine :

`contextmanager.__enter__()`

Entre dans le contexte à l'exécution, soit se renvoyant lui-même, soit en renvoyant un autre objet en lien avec ce contexte. La valeur renvoyée par cette méthode est liée à l'identifiant donné au `as` de l'instruction `with` utilisant ce gestionnaire de contexte.

Un exemple de gestionnaire de contexte se renvoyant lui-même est *file object*. Les *file objects* se renvoient eux-même depuis `__enter__()` et autorisent `open()` à être utilisé comme contexte à une instruction `with`.

Un exemple de gestionnaire de contexte renvoyant un objet connexe est celui renvoyé par *decimal.localcontext()*. Ces gestionnaires remplacent le contexte décimal courant par une copie de l'original, copie qui est renvoyée. Ça permet de changer le contexte courant dans le corps du `with` sans affecter le code en dehors de l'instruction `with`.

`contextmanager.__exit__(exc_type, exc_val, exc_tb)`

Sort du contexte et renvoie un booléen indiquant si une exception survenue doit être supprimée. Si une exception est survenue lors de l'exécution du corps de l'instruction `with`, les arguments contiennent le type de l'exception, sa valeur, et la trace de la pile (*traceback*). Sinon les trois arguments valent `None`.

L'instruction `with` inhibera l'exception si cette méthode renvoie une valeur vraie, l'exécution continuera ainsi à l'instruction suivant immédiatement l'instruction `with`. Sinon, l'exception continuera de se propager après la fin de cette méthode. Les exceptions se produisant pendant l'exécution de cette méthode remplaceront toute exception qui s'est produite dans le corps du `with`.

L'exception reçue ne doit jamais être relancée explicitement, cette méthode devrait plutôt renvoyer une valeur fausse pour indiquer que son exécution s'est terminée avec succès et qu'elle ne veut pas supprimer l'exception. Ceci permet au code de gestion du contexte de comprendre si une méthode `__exit__()` a échoué.

Python définit plusieurs gestionnaires de contexte pour faciliter la synchronisation des fils d'exécution, la fermeture des fichiers ou d'autres objets, et la configuration du contexte arithmétique décimal. Ces types spécifiques ne sont pas traités différemment, ils respectent simplement le protocole de gestion du contexte. Voir les exemples dans la documentation du module `contextlib`.

Les *generators* de Python et le décorateur `contextlib.contextmanager` permettent d'implémenter simplement ces protocoles. Si un générateur est décoré avec `contextlib.contextmanager`, elle renverra un gestionnaire de

contexte implémentant les méthodes `__enter__()` et `__exit__()`, plutôt que l'itérateur produit par un générateur non décoré.

Notez qu'il n'y a pas d'emplacement spécifique pour ces méthodes dans la structure de type pour les objets Python dans l'API Python/C. Les types souhaitant définir ces méthodes doivent les fournir comme une méthode accessible en Python. Comparé au coût de la mise en place du contexte d'exécution, le coût d'un accès au dictionnaire d'une classe unique est négligeable.

4.12 Autres types natifs

L'interpréteur gère aussi d'autres types d'objets, la plupart ne supportant cependant qu'une ou deux opérations.

4.12.1 Modules

La seule opération spéciale sur un module est l'accès à ses attributs : `m.name`, où *m* est un module et *name* donne accès un nom défini dans la table des symboles de *m*. Il est possible d'assigner un attribut de module. (Notez que l'instruction `import` n'est pas strictement une opération sur un objet module. `import foo` ne nécessite pas qu'un objet module nommé *foo* existe, il nécessite cependant une *définition* (externe) d'un module nommé *foo* quelque part.)

Un attribut spécial à chaque module est `__dict__`. C'est le dictionnaire contenant la table des symboles du module. Modifier ce dictionnaire changera la table des symboles du module, mais assigner directement `__dict__` n'est pas possible (vous pouvez écrire `m.__dict__['a'] = 1`, qui donne 1 comme valeur pour `m.a`, mais vous ne pouvez pas écrire `m.__dict__ = {}`). Modifier `__dict__` directement n'est pas recommandé.

Les modules natifs à l'interpréteur sont représentés `<module 'sys' (built-in)>`. S'ils sont chargés depuis un fichier, ils sont représentés `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

4.12.2 Les classes et instances de classes

Voir `objects` et `class`.

4.12.3 Fonctions

Les objets fonctions sont créés par les définitions de fonctions. La seule opération applicable à un objet fonction est de l'appeler : `func(argument-list)`.

Il existe en fait deux catégories d'objets fonctions : Les fonctions natives et les fonctions définies par l'utilisateur. Les deux gèrent les mêmes opérations (l'appel à la fonction), mais leur implémentation est différente, d'où les deux types distincts.

Voir `function` pour plus d'information.

4.12.4 Méthodes

Les méthodes sont des fonctions appelées via la notation d'attribut. Il en existe deux variantes : Les méthodes natives (tel que `append()` sur les listes), et les méthodes d'instances de classes. Les méthodes natives sont représentées avec le type qui les supporte.

Si vous accédez à une méthode (une fonction définie dans l'espace de nommage d'une classe) via une instance, vous obtenez un objet spécial, une *bound method* (aussi appelée *instance method*). Lorsqu'elle est appelée, elle ajoute l'argument `self` à la liste des arguments. Les méthodes liées ont deux attributs spéciaux, en lecture seule : `m.__self__` est l'objet sur

lequel la méthode travaille, et `m.__func__` est la fonction implémentant la méthode. Appeler `m(arg-1, arg-2, ..., arg-n)` est tout à fait équivalent à appeler `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Comme les objets fonctions, les objets méthodes, liées, acceptent des attributs arbitraires. Cependant, puisque les attributs de méthodes doivent être stockés dans la fonction sous-jacente (`meth.__func__`), affecter des attributs à des objets *bound method* est interdit. Toute tentative d'affecter un attribut sur un objet *bound method* lèvera une `AttributeError`. Pour affecter l'attribut, vous devrez explicitement l'affecter à sa fonction sous-jacente :

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method' # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

Voir types pour plus d'information.

4.12.5 Objets code

Les objets code sont utilisés par l'implémentation pour représenter du code Python « pseudo-compilé », comme un corps de fonction. Ils sont différents des objets fonction dans le sens où ils ne contiennent pas de référence à leur environnement global d'exécution. Les objets code sont renvoyés par la fonction native `compile()` et peuvent être obtenus des objets fonction via leur attribut `__code__`. Voir aussi le module `code`.

Les objets code peuvent être exécutés ou évalués en les passant (au lieu d'une chaîne contenant du code) aux fonction natives `exec()` ou `eval()`.

Voir types pour plus d'information.

4.12.6 Objets type

Les objets types représentent les différents types d'objets. Le type d'un objet est obtenu via la fonction native `type()`. Il n'existe aucune opération spéciale sur les types. Le module standard `types` définit les noms de tous les types natifs.

Les types sont représentés : `<class 'int'>`.

4.12.7 L'objet Null

Cet objet est renvoyé par les fonctions ne renvoyant pas explicitement une valeur. Il ne supporte aucune opération spéciale. Il existe exactement un objet *null* nommé `None` (c'est un nom natif). `type(None)()`.

C'est écrit `None`.

4.12.8 L'objet points de suspension

Cet objet est utilisé classiquement lors des découpes (voir *slicings*). Il ne supporte aucune opération spéciale. Il n'y a qu'un seul objet *ellipsis*, nommé *Ellipsis* (un nom natif). `type(Ellipsis)()` produit le *singleton* *Ellipsis*.

C'est écrit *Ellipsis* ou `...`.

4.12.9 L'objet *NotImplemented*

Cet objet est renvoyé depuis des comparaisons ou des opérations binaires effectuées sur des types qu'elles ne supportent pas. Voir comparaisons pour plus d'informations. Il n'y a qu'un seul objet *NotImplemented*. `type(NotImplemented)()` renvoie un *singleton*.

C'est écrit *NotImplemented*.

4.12.10 Valeurs booléennes

Les valeurs booléennes sont les deux objets constants *False* et *True*. Ils sont utilisés pour représenter les valeurs de vérité (bien que d'autres valeurs peuvent être considérées vraies ou fausses). Dans des contextes numériques (par exemple en argument d'un opérateur arithmétique), ils se comportent comme les nombres entiers 0 et 1, respectivement. La fonction native *bool()* peut être utilisée pour convertir n'importe quelle valeur en booléen tant que la valeur peut être interprétée en une valeur de vérité (voir *Valeurs booléennes* au dessus).

Ils s'écrivent *False* et *True*, respectivement.

4.12.11 Objets internes

Voir types. Ils décrivent les objets *stack frame*, *traceback*, et *slice*.

4.13 Attributs spéciaux

L'implémentation ajoute quelques attributs spéciaux et en lecture seule, à certains types, lorsque ça a du sens. Certains ne sont *pas* listés par la fonction native *dir()*.

`object.__dict__`

Un dictionnaire ou un autre *mapping object* utilisé pour stocker les attributs (modifiables) de l'objet.

`instance.__class__`

La classe de l'instance de classe.

`class.__bases__`

Le *tuple* des classes parentes d'un objet classe.

`definition.__name__`

Le nom de la classe, fonction, méthode, descripteur, ou générateur.

`definition.__qualname__`

Le *qualified name* de la classe, fonction, méthode, descripteur, ou générateur.

Nouveau dans la version 3.3.

`class.__mro__`

Cet attribut est un *tuple* contenant les classes parents prises en compte lors de la résolution de méthode.

`class.mro()`

Cette méthode peut être surchargée par une méta-classe pour personnaliser l'ordre de la recherche de méthode pour ses instances. Elle est appelée à la l'initialisation de la classe, et son résultat est stocké dans l'attribut `__mro__`.

`class.__subclasses__()`

Chaque classe garde une liste de références faibles à ses classes filles immédiates. Cette méthode renvoie la liste de toutes ces références encore valables. Exemple :

```
>>> int.__subclasses__()
[<class 'bool'>]
```

Notes

Exceptions natives

En python, une exception est une instance d'une classe héritée de `BaseException`. Dans un bloc `try`, la clause `except` traite non seulement la classe d'exception qu'elle mentionne, mais aussi toutes les classes dérivées de cette classe (contrairement à ses classes mères). Deux classes qui ne sont pas liées par héritage ne sont jamais équivalentes, même si elles ont le même nom.

Les exceptions natives présentes ci-dessous peuvent être levées par l'interpréteur ou par les fonctions natives. Sauf mention contraire, une « valeur associée » indique la cause de l'erreur. Cela peut être une chaîne ou un *tuple* contenant plusieurs éléments d'information (e.g., un code d'erreur ou un message explicatif). Cette valeur associée est généralement donnée en argument du constructeur de la classe.

Du code utilisateur peut lever des exceptions natives. Cela peut être utilisé pour tester un gestionnaire d'exception ou pour rapporter une condition d'erreur « comme si » c'était l'interpréteur qui levait cette exception ; mais attention car rien n'empêche du code utilisateur de lever une erreur inappropriée.

Les classes d'exception natives peuvent être héritées pour définir de nouvelles exceptions ; les programmeurs sont encouragés à faire dériver les nouvelles exceptions de la classe `Exception` ou d'une de ses sous-classes, et non de `BaseException`. Plus d'informations sur la définition des exceptions sont disponibles dans le Tutoriel Python sous `tut-userexceptions`.

En levant (ou levant à nouveau) une exception dans une clause `except` ou `finally`, `__context__` est automatiquement assigné à la dernière exception capturée ; si la nouvelle exception n'est pas gérée, la trace d'appels affichée inclut la ou les exception(s) d'origine et l'exception finale.

En levant une nouvelle exception (plutôt que d'utiliser un simple `raise` pour lever à nouveau l'exception en cours de traitement), le contexte implicite d'exception peut être complété par une cause explicite en utilisant `from` avec `raise` :

```
raise new_exc from original_exc
```

The expression following `from` must be an exception or `None`. It will be set as `__cause__` on the raised exception. Setting `__cause__` also implicitly sets the `__suppress_context__` attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes (e.g. converting `KeyError` to `AttributeError`), while leaving the old exception available in `__context__` for introspection when debugging.

Le code d'affichage par défaut de la trace d'appels montre ces exceptions chaînées en plus de la trace de l'exception

elle-même. Une exception chaînée explicitement dans `__cause__` est toujours affichée si présente. Une exception implicitement chaînée dans `__context__` n'est affichée que si `__cause__` est *None* et `__suppress_context__` est faux.

Dans les deux cas, l'exception elle-même est toujours affichée après toutes les exceptions enchaînées, de sorte que la dernière ligne de la trace d'appels montre toujours la dernière exception qui a été levée.

5.1 Classes de base

Les exceptions suivantes sont utilisées principalement en tant que classes de base pour d'autres exceptions.

exception BaseException

La classe de base pour toutes les exceptions natives. Elle n'est pas vouée à être héritée directement par des classes utilisateur (pour cela, utilisez *Exception*). Si *str()* est appelée sur une instance de cette classe, la représentation du ou des argument(s) de l'instance est retournée, ou la chaîne vide s'il n'y avait pas d'arguments.

args

Le *tuple* d'arguments donné au constructeur d'exception. Certaines exceptions natives (comme *OSError*) attendent un certain nombre d'arguments et attribuent une signification spéciale aux éléments de ce *tuple*, alors que d'autres ne sont généralement appelées qu'avec une seule chaîne de caractères rendant un message d'erreur.

with_traceback (tb)

Cette méthode définit *tb* en tant que nouvelle trace d'appels pour l'exception et retourne l'objet exception. Elle est généralement utilisée dans du code de gestion d'exceptions comme ceci :

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

exception Exception

Toutes les exceptions natives, qui n'entraînent pas une sortie du système dérivent de cette classe. Toutes les exceptions définies par l'utilisateur devraient également être dérivées de cette classe.

exception ArithmeticError

La classe de base pour les exceptions natives qui sont levées pour diverses erreurs arithmétiques : *OverflowError*, *ZeroDivisionError*, *FloatingPointError*.

exception BufferError

Levée lorsqu'une opération liée à un tampon ne peut pas être exécutée.

exception LookupError

La classe de base pour les exceptions qui sont levées lorsqu'une clé ou un index utilisé sur un tableau de correspondances ou une séquence est invalide : *IndexError*, *KeyError*. Peut être levée directement par *codecs.lookup()*.

5.2 Exceptions concrètes

Les exceptions suivantes sont celles qui sont habituellement levées.

exception `AssertionError`

Levée lorsqu'une instruction `assert` échoue.

exception `AttributeError`

Levée lorsqu'une référence ou une assignation d'attribut (voir `attribute-references`) échoue. (Lorsqu'un objet ne supporte pas du tout la référence ou l'assignation d'attribut, `TypeError` est levé.)

exception `EOFError`

Levée lorsque la fonction `input()` atteint une condition de fin de fichier (EOF) sans lire aucune donnée. (N.B. : les méthodes `io.IOBase.read()` et `io.IOBase.readline()` retournent une chaîne vide lorsqu'elles atteignent EOF.)

exception `FloatingPointError`

Levée lorsqu'une opération en virgule flottante échoue. Cette exception est toujours définie, mais ne peut être levée que lorsque Python est configuré avec l'option `--with-fpectl`, ou que le symbole `WANT_SIGFPE_HANDLER` est défini dans le fichier `pyconfig.h`.

exception `GeneratorExit`

Levée lorsqu'un *generator* ou une *coroutine* est fermé, voir `generator.close()` et `coroutine.close()`. Elle hérite directement de `BaseException` au lieu de `Exception` puisqu'il ne s'agit pas techniquement d'une erreur.

exception `ImportError`

Levée lorsque l'instruction `import` a des problèmes pour essayer de charger un module. Également levée lorsque Python ne trouve pas un nom dans `from ... import`.

Les attributs `name` et `path` peuvent être définis uniquement à l'aide d'arguments mot-clé (*kwargs*) passés au constructeur. Lorsqu'ils sont définis, ils représentent respectivement le nom du module qui a été tenté d'être importé et le chemin d'accès au fichier qui a déclenché l'exception.

Modifié dans la version 3.3 : Ajout des attributs `name` et `path`.

exception `ModuleNotFoundError`

Une sous-classe de `ImportError` qui est levée par `import` lorsqu'un module n'a pas pu être localisé. Elle est généralement levée quand `None` est trouvé dans `sys.modules`.

Nouveau dans la version 3.6.

exception `IndexError`

Levée lorsqu'un indice de séquence est hors de la plage. (Les indices de tranches (*slices*) sont tronqués silencieusement pour tomber dans la plage autorisée ; si un indice n'est pas un entier, `TypeError` est levée.)

exception `KeyError`

Levée lorsqu'une clef (de dictionnaire) n'est pas trouvée dans l'ensemble des clefs existantes.

exception `KeyboardInterrupt`

Levée lorsque l'utilisateur appuie sur la touche d'interruption (normalement `Control-C` or `Delete`). Pendant l'exécution, un contrôle des interruptions est effectué régulièrement. L'exception hérite de `BaseException` afin de ne pas être accidentellement capturée par du code qui capture `Exception` et ainsi empêcher l'interpréteur de quitter.

exception `MemoryError`

Levée lorsqu'une opération est à court de mémoire mais que la situation peut encore être rattrapée (en supprimant certains objets). La valeur associée est une chaîne de caractères indiquant quel type d'opération (interne) est à court de mémoire. À noter qu'en raison de l'architecture interne de gestion de la mémoire (la fonction `malloc()` du C), l'interpréteur peut ne pas toujours être capable de rattraper cette situation ; il lève néanmoins une exception pour qu'une pile d'appels puisse être affichée, dans le cas où un programme en cours d'exécution en était la cause.

exception NameError

Levée lorsqu'un nom local ou global n'est pas trouvé. Ceci ne s'applique qu'aux noms non qualifiés. La valeur associée est un message d'erreur qui inclut le nom qui n'a pas pu être trouvé.

exception NotImplementedError

Cette exception est dérivée de *RuntimeError*. Dans les classes de base définies par l'utilisateur, les méthodes abstraites devraient lever cette exception lorsqu'elles nécessitent des classes dérivées pour remplacer la méthode, ou lorsque la classe est en cours de développement pour indiquer que l'implémentation concrète doit encore être ajoutée.

Note : Elle ne devrait pas être utilisée pour indiquer qu'un opérateur ou qu'une méthode n'est pas destiné à être pris en charge du tout – dans ce cas, laissez soit l'opérateur / la méthode non défini, soit, s'il s'agit d'une sous-classe, assignez-le à *None*.

Note : *NotImplementedError* et *NotImplemented* ne sont pas interchangeables, même s'ils ont des noms et des objectifs similaires. Voir *NotImplemented* pour des détails sur la façon de les utiliser.

exception OSError ([arg])**exception OSError (errno, strerror[, filename[, winerror[, filename2]]])**

Cette exception est levée lorsqu'une fonction système retourne une erreur liée au système, incluant les erreurs entrées-sorties telles que « fichier non trouvé » ou « disque plein » (pas pour les types d'arguments illégaux ou d'autres erreurs accidentelles).

La deuxième forme du constructeur définit les attributs correspondants, décrits ci-dessous. Les attributs par défaut sont *None* si non spécifiés. Pour la rétrocompatibilité, si trois arguments sont passés, l'attribut *args* contient seulement un *tuple* à deux valeurs des deux premiers arguments du constructeur.

Le constructeur retourne souvent une sous-classe d'*OSError*, comme décrit dans *OS exceptions* ci-dessous. La sous-classe particulière dépend de la valeur finale de *errno*. Ce comportement ne se produit que lors de la construction d'*OSError* directement ou via un alias, et n'est pas hérité lors du sous-classement.

errno

Code d'erreur numérique de la variable C *errno*.

winerror

Sous Windows, cela donne le code d'erreur Windows natif. L'attribut *errno* est alors une traduction approximative, en termes POSIX, de ce code d'erreur natif.

Sous Windows, si l'argument du constructeur *winerror* est un entier, l'attribut *errno* est déterminé à partir du code d'erreur Windows, et l'argument *errno* est ignoré. Sur d'autres plateformes, l'argument *winerror* est ignoré, et l'attribut *winerror* n'existe pas.

strerror

Le message d'erreur correspondant, tel que fourni par le système d'exploitation. Il est formaté par les fonctions *C perror()* sous POSIX, et *FormatMessage()* sous Windows.

filename**filename2**

Pour les exceptions qui font référence à un chemin d'accès au système de fichiers (comme *open()* ou *os.unlink()*), *filename* est le nom du fichier transmis à la fonction. Pour les fonctions qui font référence à deux chemins d'accès au système de fichiers (comme *os.rename()*), *filename2* correspond au deuxième nom de fichier passé à la fonction.

Modifié dans la version 3.3 : *EnvironmentError*, *IOError*, *WindowsError*, *socket.error*, *select.error* et *mmap.error* ont fusionnées en *OSError*, et le constructeur peut renvoyer une sous-classe.

Modifié dans la version 3.4 : L'attribut *filename* est maintenant le nom du fichier originel passé à la fonction, au lieu du nom encodé ou décodé à partir de l'encodage du système de fichiers. De plus, l'argument du constructeur et l'attribut *filename2* a été ajouté.

exception OverflowError

Levée lorsque le résultat d'une opération arithmétique est trop grand pour être représenté. Cela ne peut pas se produire pour les entiers (qui préfèrent lever `MemoryError` plutôt que d'abandonner). Cependant, pour des raisons historiques, `OverflowError` est parfois levée pour des entiers qui sont en dehors d'une plage requise. En raison de l'absence de normalisation de la gestion des exceptions de virgule flottante en C, la plupart des opérations en virgule flottante ne sont pas vérifiées.

exception RecursionError

Cette exception est dérivée de `RuntimeError`. Elle est levée lorsque l'interpréteur détecte que la profondeur de récursivité maximale (voir `sys.getrecursionlimit()`) est dépassée.

Nouveau dans la version 3.5 : Auparavant, une simple `RuntimeError` était levée.

exception ReferenceError

Cette exception est levée lorsqu'un pointeur faible d'un objet proxy, créé par la fonction `weakref.proxy()`, est utilisé pour accéder à un attribut du référent après qu'il ait été récupéré par le ramasse-miettes. Pour plus d'informations sur les pointeurs faibles, voir le module `weakref`.

exception RuntimeError

Levée lorsqu'une erreur qui n'appartient à aucune des autres catégories est détectée. La valeur associée est une chaîne de caractères indiquant précisément ce qui s'est mal passé.

exception StopIteration

Levée par la fonction native `next()` et la méthode `__next__()` d'un *iterator* (itérateur) pour signaler qu'il n'y a pas d'autres éléments produits par l'itérateur.

L'objet exception a un unique attribut `value`, qui est donné en argument lors de la construction de l'exception, et vaut `None` par défaut.

Lorsqu'une fonction de type *generator* ou *coroutine* retourne une valeur, une nouvelle instance de `StopIteration` est levée, et la valeur retournée par la fonction est passée au paramètre `value` du constructeur de l'exception.

Si une fonction de générateur définit en présence d'une directive `from __future__ import generator_stop` lève `StopIteration`, elle est convertie en `RuntimeError` (en conservant `StopIteration` comme cause de la nouvelle exception).

Modifié dans la version 3.3 : Ajout de l'attribut `value` et de la possibilité pour les fonctions de générateur de l'utiliser pour retourner une valeur.

Modifié dans la version 3.5 : Introduction de la transformation `RuntimeError`.

exception StopAsyncIteration

Doit être levée par la méthode `__anext__()` d'un objet *asynchronous iterator* pour arrêter l'itération.

Nouveau dans la version 3.5.

exception SyntaxError

Levée lorsque l'analyseur syntaxique rencontre une erreur de syntaxe. Cela peut se produire dans une instruction `import`, dans un appel aux fonctions natives `exec()` ou `eval()`, ou lors de la lecture du script initial ou de l'entrée standard (également de manière interactive).

Les instances de cette classe ont des attributs `filename`, `lineno`, `offset` et `text` pour accéder plus facilement aux détails. La représentation `str()` de l'instance de l'exception retourne seulement le message.

exception IndentationError

Classe de base pour les erreurs de syntaxe liées à une indentation incorrecte. C'est une sous-classe de `SyntaxError`.

exception TabError

Levée lorsqu'une indentation contient une utilisation incohérente des tabulations et des espaces. C'est une sous-classe de `IndentationError`.

exception SystemError

Levée lorsque l'interpréteur trouve une erreur interne, mais que la situation ne semble pas si grave au point de lui faire abandonner tout espoir. La valeur associée est une chaîne de caractères indiquant l'erreur qui est survenue (en termes bas niveau).

Vous devriez le signaler à l'auteur ou au responsable de votre interpréteur Python. Assurez-vous de signaler la version de l'interpréteur (`sys.version`; elle est également affichée au lancement d'une session interactive), le message d'erreur exact (la valeur associée à l'exception) et si possible le code source du programme qui a déclenché l'erreur.

exception SystemExit

Cette exception est levée par la fonction `sys.exit()`. Elle hérite de `BaseException` au lieu d'`Exception` pour ne pas qu'elle soit accidentellement capturée par du code qui capture `Exception`. Cela permet à l'exception de se propager correctement et de faire quitter l'interpréteur. Lorsqu'elle n'est pas gérée, l'interpréteur Python quitte; aucune trace d'appels n'est affichée. Le constructeur accepte le même argument optionnel passé à `sys.exit()`. Si la valeur est un entier, elle spécifie l'état de sortie du système (passé à la fonction `C exit()`); si elle est `None`, l'état de sortie est zéro; si elle a un autre type (comme une chaîne de caractères), la valeur de l'objet est affichée et l'état de sortie est un.

Un appel à `sys.exit()` est traduit en une exception pour que les gestionnaires de nettoyage (les clauses `finally` des instructions `try`) puissent être exécutés, et pour qu'un débogueur puisse exécuter un script sans courir le risque de perdre le contrôle. La fonction `os._exit()` peut être utilisée s'il est absolument nécessaire de sortir immédiatement (par exemple, dans le processus enfant après un appel à `os.fork()`).

code

L'état de sortie ou le message d'erreur passé au constructeur. (`None` par défaut.)

exception TypeError

Levée lorsqu'une opération ou fonction est appliquée à un objet d'un type inapproprié. La valeur associée est une chaîne de caractères donnant des détails sur le type d'inadéquation.

Cette exception peut être levée par du code utilisateur pour indiquer qu'une tentative d'opération sur un objet n'est pas prise en charge, et n'est pas censée l'être. Si un objet est destiné à prendre en charge une opération donnée mais n'a pas encore fourni une implémentation, lever `NotImplementedError` est plus approprié.

Le passage d'arguments du mauvais type (e.g. passer une `list` quand un `int` est attendu) devrait résulter en un `TypeError`, mais le passage d'arguments avec la mauvaise valeur (e.g. un nombre en dehors des limites attendues) devrait résulter en une `ValueError`.

exception UnboundLocalError

Levée lorsqu'une référence est faite à une variable locale dans une fonction ou une méthode, mais qu'aucune valeur n'a été liée à cette variable. C'est une sous-classe de `NameError`.

exception UnicodeError

Levée lorsqu'une erreur d'encodage ou de décodage liée à Unicode se produit. C'est une sous-classe de `ValueError`.

`UnicodeError` a des attributs qui décrivent l'erreur d'encodage ou de décodage. Par exemple, `err.object[err.start:err.end]` donne l'entrée particulière invalide sur laquelle le codec a échoué.

encoding

Le nom de l'encodage qui a provoqué l'erreur.

reason

Une chaîne de caractères décrivant l'erreur de codec spécifique.

object

L'objet que le codec essayait d'encoder ou de décoder.

start

Le premier index des données invalides dans `object`.

end

L'index après la dernière donnée invalide dans `object`.

exception UnicodeEncodeError

Levée lorsqu'une erreur liée à Unicode se produit durant l'encodage. C'est une sous-classe d'`UnicodeError`.

exception UnicodeDecodeError

Levée lorsqu'une erreur liée à Unicode se produit durant le décodage. C'est une sous-classe d'`UnicodeError`.

exception UnicodeTranslateError

Levée lorsqu'une erreur liée à Unicode se produit durant la traduction. C'est une sous-classe d'*UnicodeError*.

exception ValueError

Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as *IndexError*.

exception ZeroDivisionError

Levée lorsque le second argument d'une opération de division ou d'un modulo est zéro. La valeur associée est une chaîne indiquant le type des opérandes et de l'opération.

Les exceptions suivantes sont conservées pour la compatibilité avec les anciennes versions ; depuis Python 3.3, ce sont des alias d'*OSError*.

exception EnvironmentError**exception IOError****exception WindowsError**

Seulement disponible sous Windows.

5.2.1 Exceptions système

Les exceptions suivantes sont des sous-classes d'*OSError*, elles sont levées en fonction du code d'erreur système.

exception BlockingIOError

Levée lorsqu'une opération bloque sur un objet (par exemple un connecteur) configuré pour une opération non-bloquante. Correspond à `errno EAGAIN`, `EALREADY`, `EWOULDBLOCK` et `EINPROGRESS`.

En plus de ceux de *OSError*, *BlockingIOError* peut avoir un attribut de plus :

characters_written

Un nombre entier contenant le nombre de caractères écrits dans le flux avant qu'il ne soit bloqué. Cet attribut est disponible lors de l'utilisation des classes tampon entrées-sorties du module *io*.

exception ChildProcessError

Levée lorsqu'une opération sur un processus enfant a échoué. Correspond à `errno ECHILD`.

exception ConnectionError

Une classe de base pour les problèmes de connexion.

Les sous-classes sont *BrokenPipeError*, *ConnectionAbortedError*, *ConnectionRefusedError* et *ConnectionResetError*.

exception BrokenPipeError

Une sous-classe de *ConnectionError*, levée en essayant d'écrire sur un *pipe* alors que l'autre extrémité a été fermée, ou en essayant d'écrire sur un connecteur (*socket* en anglais) qui a été fermé pour l'écriture. Correspond à `errno EPIPE` et `ESHUTDOWN`.

exception ConnectionAbortedError

Une sous-classe de *ConnectionError*, levée lorsqu'une tentative de connexion est interrompue par le pair. Correspond à `errno ECONNABORTED`.

exception ConnectionRefusedError

Une sous-classe de *ConnectionError*, levée lorsqu'une tentative de connexion est refusée par le pair. Correspond à `errno ECONNREFUSED`.

exception ConnectionResetError

Une sous-classe de *ConnectionError*, levée lorsqu'une connexion est réinitialisée par le pair. Correspond à `errno ECONNRESET`.

exception FileExistsError

Levée en essayant de créer un fichier ou un répertoire qui existe déjà. Correspond à `errno EEXIST`.

exception FileNotFoundError

Levée lorsqu'un fichier ou répertoire est demandé mais n'existe pas. Correspond à `errno` `ENOENT`.

exception InterruptedError

Levée lorsqu'un appel système est interrompu par un signal entrant. Correspond à `errno` `EINTR`.

Modifié dans la version 3.5 : Python relance maintenant les appels système lorsqu'ils sont interrompus par un signal, sauf si le gestionnaire de signal lève une exception (voir [PEP 475](#) pour les raisons), au lieu de lever `InterruptedError`.

exception IsADirectoryError

Levée lorsqu'une opération sur un fichier (comme `os.remove()`) est demandée sur un répertoire. Correspond à `errno` `EISDIR`.

exception NotADirectoryError

Levée lorsqu'une opération sur un répertoire (comme `os.listdir()`) est demandée sur autre chose qu'un répertoire. Correspond à `errno` `ENOTDIR`.

exception PermissionError

Levée lorsqu'on essaye d'exécuter une opération sans les droits d'accès adéquats — par exemple les permissions du système de fichiers. Correspond à `errno` `EACCES` et `EPERM`.

exception ProcessLookupError

Levée lorsqu'un processus donné n'existe pas. Correspond à `errno` `ESRCH`.

exception TimeoutError

Levée lorsqu'une fonction système a expiré au niveau système. Correspond à `errno` `ETIMEDOUT`.

Nouveau dans la version 3.3 : Toutes les sous-classes d'`OSError` ci-dessus ont été ajoutées.

Voir aussi :

[PEP 3151](#) – Refonte de la hiérarchie des exceptions système et IO

5.3 Avertissements

Les exceptions suivantes sont utilisées comme catégories d'avertissement ; voir le module `warnings` pour plus d'informations.

exception Warning

Classe de base pour les catégories d'avertissement.

exception UserWarning

Classe de base pour les avertissements générés par du code utilisateur.

exception DeprecationWarning

Classe de base pour les avertissements sur les fonctionnalités obsolètes.

exception PendingDeprecationWarning

Classe de base pour les avertissements sur les fonctionnalités qui seront obsolètes dans le futur.

exception SyntaxWarning

Classe de base pour les avertissements sur de la syntaxe douteuse.

exception RuntimeWarning

Classe de base pour les avertissements sur les comportements d'exécution douteux.

exception FutureWarning

Classe de base pour les avertissements sur les constructions qui changeront sémantiquement dans le futur.

exception ImportWarning

Classe de base pour les avertissements sur des erreurs probables dans les importations de modules.

exception UnicodeWarning

Classe de base pour les avertissements liés à l'Unicode.

exception BytesWarning

Classe de base pour les avertissements liés à *bytes* et *bytearray*.

exception ResourceWarning

Classe de base pour les avertissements liés à l'utilisation de ressources.

Nouveau dans la version 3.2.

5.4 Hiérarchie des exceptions

La hiérarchie de classes pour les exceptions natives est la suivante :

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |   +-- ModuleNotFoundError
    +-- LookupError
    |   +-- IndexError
    |   +-- KeyError
    +-- MemoryError
    +-- NameError
    |   +-- UnboundLocalError
    +-- OSError
    |   +-- BlockingIOError
    |   +-- ChildProcessError
    |   +-- ConnectionError
    |       +-- BrokenPipeError
    |       +-- ConnectionAbortedError
    |       +-- ConnectionRefusedError
    |       +-- ConnectionResetError
    |   +-- FileExistsError
    |   +-- FileNotFoundError
    |   +-- InterruptedError
    |   +-- IsADirectoryError
    |   +-- NotADirectoryError
    |   +-- PermissionError
    |   +-- ProcessLookupError
    |   +-- TimeoutError
    +-- ReferenceError
    +-- RuntimeError
    |   +-- NotImplementedError
```

(suite sur la page suivante)

(suite de la page précédente)

```
|    +-- RecursionError
+-- SyntaxError
|    +-- IndentationError
|        +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|        +-- UnicodeDecodeError
|        +-- UnicodeEncodeError
|        +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```

Services de Manipulation de Texte

Les modules décrits dans ce chapitre fournissent un large ensemble d'opérations de manipulation sur les chaînes de caractères et le texte en général.

Le module *codecs* documenté dans *Services autour des Données Binaires* est aussi très pertinent pour la manipulation de texte. Consultez aussi la documentation du type *str* natif Python dans *Type Séquence de Texte — str*.

6.1 string — Opérations usuelles sur des chaînes

Code source : [Lib/string.py](#)

Voir aussi :

Type Séquence de Texte — str

Méthodes de chaînes de caractères

6.1.1 Chaînes constantes

Les constantes définies dans ce module sont :

`string.ascii_letters`

La concaténation des constantes *ascii_lowercase* et *ascii_uppercase* décrites ci-dessous. Cette valeur n'est pas dépendante de l'environnement linguistique.

`string.ascii_lowercase`

Les lettres minuscules 'abcdefghijklmnopqrstuvwxyz'. Cette valeur de dépend pas de l'environnement linguistique et ne changera pas.

`string.ascii_uppercase`

Les lettres majuscules 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. Cette valeur de dépend pas de l'environnement linguistique et ne changera pas.

`string.digits`

La chaîne '0123456789'.

`string.hexdigits`

La chaîne '0123456789abcdefABCDEF'.

`string.octdigits`

La chaîne '01234567'.

`string.punctuation`

Chaîne de caractères ASCII considérés comme ponctuation dans l'environnement linguistique C.

`string.printable`

Chaîne de caractères ASCII considérés comme affichables. C'est une combinaison de *digits*, *ascii_letters*, *punctuation*, et *whitespace*.

`string.whitespace`

Une chaîne comprenant tous les caractères ASCII considérés comme espaces. Sont inclus les caractères espace, tabulations, saut de ligne, retour du chariot, saut de page, et tabulation verticale.

6.1.2 Formatage personnalisé de chaîne

La classe primitive *string* fournit la possibilité de faire des substitutions de variables complexes et du formatage de valeurs via la méthode *format()* décrite par **PEP 3101**. La classe *Formatter* dans le module *string* permet de créer et personnaliser vos propres comportements de formatage de chaînes en utilisant la même implémentation que la méthode primitive *format()*.

class `string.Formatter`

La classe *Formatter* a les méthodes publiques suivantes :

format (*format_string*, **args*, ***kwargs*)

La méthode principale de l'API. Elle prend une chaîne de format et un ensemble arbitraire d'arguments positions et mot-clefs. C'est uniquement un conteneur qui appelle *vformat()*.

Obsolète depuis la version 3.5 : Passer la chaîne de format comme argument mot-clef *format_string* est obsolète.

vformat (*format_string*, *args*, *kwargs*)

Cette fonction fait le travail effectif du formatage. Elle existe comme méthode séparée au cas où vous voudriez passer un dictionnaire d'arguments prédéfini plutôt que décompresser et recompresser le dictionnaire en arguments individuels en utilisant la syntaxe **args* et ***kwargs*. *vformat()* s'occupe de découper la chaîne de format en données de caractères et champs de remplacement. Elle appelle les différentes méthodes décrites ci-dessous.

De plus, la classe *Formatter* définit un certain nombre de méthodes qui ont pour vocation d'être remplacées par des sous-classes :

parse (*format_string*)

Boucle sur la chaîne de format et renvoie un itérable de *tuples* (*literal_text*, *field_name*, *format_spec*, *conversion*). Ceci est utilisé par *vformat()* pour découper la chaîne de format en littéraux ou en champs de remplacement.

Les valeurs dans le *tuple* représentent conceptuellement un ensemble de littéraux suivis d'un unique champ de remplacement. S'il n'y a pas de littéral, (ce qui peut arriver si deux champs de remplacement sont placés côte à côte), alors *literal_text* est une chaîne vide. S'il n'y a pas de champ de remplacement, les valeurs *field_name*, *format_spec* et *conversion* sont mises à *None*.

get_field (*field_name*, *args*, *kwargs*)

Récupère le champ *field_name* du *tuple* renvoyé par *parse()* (voir ci-dessus), le convertit en un objet à formater. Renvoie un *tuple* (*obj*, *used_key*). La version par défaut prend une chaîne de la forme définie par **PEP 3101**, telle que « *0[name]* » ou « *label.title* ». *args* et *kwargs* sont tels que ceux passés à *vformat()*. La valeur renvoyée *used_key* a le même sens que le paramètre *key* de *get_value()*.

get_value (*key*, *args*, *kwargs*)

Récupère la valeur d'un champ donné. L'argument *key* est soit un entier, soit une chaîne. Si c'est un entier, il représente l'indice de l'argument dans *args*. Si c'est une chaîne de caractères, elle représente le nom de l'argument dans *kwargs*.

Le paramètre *args* est défini par la liste des arguments positionnels de `vformat()`, et le paramètre *kwargs* est défini par le dictionnaire des arguments mot-clefs.

Pour les noms de champs composés, ces fonctions sont uniquement appelées sur la première composante du nom. Les composantes suivantes sont manipulées au travers des attributs normaux et des opérations sur les indices.

Donc par exemple, le champ-expression `0.name` amènerait `get_value()` à être appelée avec un argument *key* valant 0. L'attribut `name` sera recherché après l'appel `get_value()` en faisant appel à la primitive `getattr()`.

Si l'indice ou le mot-clef fait référence à un objet qui n'existe pas, alors une exception `IndexError` ou `KeyError` doit être levée.

check_unused_args (*used_args*, *args*, *kwargs*)

Implémente une vérification pour les arguments non utilisés si désiré. L'argument de cette fonction est l'ensemble des clefs qui ont été effectivement référencées dans la chaîne de format (des entiers pour les indices et des chaînes de caractères pour les arguments nommés), et une référence vers les arguments *args* et *kwargs* qui ont été passés à `vformat`. L'ensemble des arguments non utilisés peut être calculé sur base de ces paramètres. `check_unused_args()` est censée lever une exception si la vérification échoue.

format_field (*value*, *format_spec*)

La méthode `format_field()` fait simplement appel à la primitive globale `format()`. Cette méthode est fournie afin que les sous-classes puisse la redéfinir.

convert_field (*value*, *conversion*)

Convertit la valeur (renvoyée par `get_field()`) selon un type de conversion donné (comme dans le tuple renvoyé par la méthode `parse()`). La version par défaut comprend "s" (*str*), "r" (*repr*) et "a" (ASCII) comme types de conversion.

6.1.3 Syntaxe de formatage de chaîne

La méthode `str.format()` et la classe `Formatter` partagent la même syntaxe pour les chaînes de formatage (bien que dans le cas de `Formatter` les sous-classes puissent définir leur propre syntaxe). La syntaxe est liée à celle des chaînes de formatage littérales, mais il y a quelques différences.

Les chaînes de formatage contiennent des « champs de remplacement » entourés d'accolades `{ }`. Tout ce qui n'est pas placé entre deux accolades est considéré comme littéral, qui est copié tel quel dans le résultat. Si vous avez besoin d'inclure une accolade en littéral, elles peuvent être échappées en les doublant : `{{ et }}`.

La grammaire pour un champ de remplacement est défini comme suit :

```
replacement_field ::= "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name         ::= arg_name ("." attribute_name | "[" element_index "]") *
arg_name           ::= [identifiant | digit+]
attribute_name     ::= identifiant
element_index      ::= digit+ | index_string
index_string       ::= <any source character except "]"> +
conversion         ::= "r" | "s" | "a"
format_spec        ::= <described in the next section>
```

En termes moins formels, un champ de remplacement peut débuter par un *field_name* qui spécifie l'objet dont la valeur va être formatée et insérée dans le résultat à l'endroit du champ de remplacement. Le *field_name* peut éventuellement être suivi d'un champ *conversion* qui est précédé d'un point d'exclamation '!', et d'un *format_spec* qui est précédé par

deux-points ' : '. Cela définit un format personnalisé pour le remplacement d'une valeur.

Voir également la section *Mini-langage de spécification de format*.

Le champ *field_name* débute par un *arg_name* qui est soit un nombre, soit un mot-clef. Si c'est un nombre, il fait référence à un des arguments positionnels et si c'est un mot-clef, il fait référence à un des arguments nommés. Si les valeurs numériques de *arg_name* dans une chaîne de format sont 0, 1, 2, ... dans l'ordre, elles peuvent être omises (toutes ou aucune), et les nombres 0, 1, 2, ... seront automatiquement insérés dans cet ordre. Puisque *arg_name* n'est pas délimité par des guillemets, il n'est pas possible de spécifier des clefs de dictionnaire arbitraires (par exemple les chaînes '10' ou ':-]') dans une chaîne de format. La valeur *arg_name* peut être suivie par un nombre d'indices ou d'expressions quelconque. Une expression de la forme '.name' sélectionne l'attribut nommé en utilisant *getattr()* alors qu'une expression de la forme '[index]' recherche l'indice en utilisant *__getitem__()*.

Modifié dans la version 3.1 : Les spécificateurs de position d'argument peuvent être omis dans les appels à *str.format()*. Donc '{ } { }'.format(a, b) est équivalent à '{0} {1}'.format(a, b).

Modifié dans la version 3.4 : Les spécificateurs de position d'argument peuvent être omis pour *Formatter*.

Quelques exemples simples de formatage de chaînes :

```
"First, thou shalt count to {0}" # References first positional argument
"Bring me a {}"                 # Implicitly references the first positional
                                # argument
"From {} to {}".format(0, 1)    # Same as "From {0} to {1}"
"My quest is {name}"            # References keyword argument 'name'
"Weight in tons {0.weight}"     # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}" # First element of keyword argument 'players'.
```

Le champ *conversion* crée une contrainte de type avant de formater. Normalement, le travail de formatage d'une valeur est fait par la méthode *__format__()* de la valeur elle-même. Cependant, dans certains cas, il est désirable de forcer un type à être formaté en une chaîne, en ré-définissant sa propre définition de formatage. En convertissant une valeur en chaîne de caractère avant d'appeler la méthode *__format__()*, on outrepassa la logique usuelle de formatage.

Actuellement, trois indicateurs sont gérés : ' !s ' qui appelle la fonction *str()* sur la valeur, ' !r ' qui appelle la fonction *repr()* et ' !a ' qui appelle la fonction *ascii()*.

Quelques exemples :

```
"Harold's a clever {0!s}"        # Calls str() on the argument first
"Bring out the holy {name!r}"    # Calls repr() on the argument first
"More {!a}"                     # Calls ascii() on the argument first
```

Le champ *format_spec* contient une spécification sur la manière selon laquelle la valeur devrait être représentée : des informations telles que la longueur du champ, l'alignement, le remplissage, la précision décimale, etc. Chaque type peut définir son propre « mini-langage de formatage » ou sa propre interprétation de *format_spec*.

La plupart des types natifs gèrent un mini-langage de formatage usuel qui est décrit dans la section suivante.

Un champ *format_spec* peut contenir un champ de remplacement imbriqué. Ces champs de remplacement imbriqués peuvent contenir un nom de champ, un indicateur de conversion, mais une imbrication récursive plus profonde n'est pas permise. Les champs de remplacement au sein de *format_spec* sont substitués avant que la chaîne *format_spec* ne soit interprétée. Cela permet que le formatage d'une valeur soit dynamiquement spécifié.

Voir la section *Exemples de formats* pour des exemples.

Mini-langage de spécification de format

Les « Spécifications de format » sont utilisées avec les champs de remplacement contenus dans les chaînes de formatage, pour définir comment les valeurs doivent être représentées (voir *Syntaxe de formatage de chaîne* et f-strings). Elles peuvent aussi être passées directement à la fonction native `format()`. Chaque type *formatable* peut définir comment les spécifications sur les valeurs de ce type doivent être interprétées.

La plupart des primitives implémentent les options suivantes, même si certaines options de formatage ne sont supportées que pour les types numériques.

Une convention généralement admise est qu'une chaîne vide ("") produit le même résultat que si vous aviez appelé `str()` sur la valeur. Une chaîne de format non vide modifie typiquement le résultat.

La forme générale d'un *spécificateur de format standard* est :

```
format_spec      ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill             ::=  <any character>
align            ::=  "<" | ">" | "=" | "^"
sign             ::=  "+" | "-" | " "
width            ::=  digit+
grouping_option  ::=  "_" | ","
precision        ::=  digit+
type             ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "
```

Si une valeur valide est spécifiée pour *align*, elle peut être précédée par un caractère *fill*, qui peut être n'importe quel caractère (espace par défaut si la valeur est omise). Il n'est pas possible d'utiliser une accolade littérale (« { » ou « } ») comme caractère *fill* dans une chaîne de formatage littérale ou avec la méthode `str.format()`. Cependant, il est possible d'insérer une accolade à l'aide d'un champ de remplacement imbriqué. Cette limitation n'affecte pas la fonction `format()`.

Le sens des différentes options d'alignement est donné comme suit :

Op-tion	Signification
'<'	Force le champ à être aligné à gauche dans l'espace disponible (c'est le spécificateur par défaut pour la plupart des objets).
'>'	Force le champ à être aligné à droite dans l'espace disponible (c'est le spécificateur par défaut pour les nombres).
'='	Force le remplissage (<i>padding</i>) à être placé après le signe (si signe il y a) mais avant les chiffres. L'option est utilisée pour afficher les champs sous la forme « +000000120 ». Cette option d'alignement est valide uniquement pour les types numériques. Elle devient activée par défaut quand « 0 » précède directement la largeur de champ.
'^'	Force le champ à être centré dans l'espace disponible.

Notons que la longueur du champ est toujours égale à la la taille nécessaire pour remplir le champ avec l'objet à moins que la valeur minimum ne soit précisée. Ainsi, si aucune valeur n'est précisée, l'option d'alignement n'a aucun sens.

L'option *sign* est uniquement valide pour les type numériques, et peut valoir :

Op-tion	Signification
'+'	indique que le signe doit être affiché pour les nombres tant positifs que négatifs.
'-'	indique que le signe doit être affiché uniquement pour les nombres négatifs (c'est le comportement par défaut).
es-pace	indique qu'un espace doit précéder les nombres positifs et qu'un signe moins doit précéder les nombres négatifs.

L'option '#' impose l'utilisation de la « forme alternative » pour la conversion. La forme alternative est définie différemment pour différents types. Cette option est uniquement valide pour les types entiers flottants, complexes, et décimaux. Pour les entiers, quand l'affichage binaire, octal ou hexadécimal est utilisé, cette option ajoute le préfixe '0b', '0o', ou '0x' à la valeur affichée. Pour les flottants, les complexes, et les décimaux, la forme alternative impose que le résultat de la conversion contienne toujours une virgule, même si aucun chiffre ne vient après. Normalement, une virgule apparaît dans le résultat de ces conversions seulement si un chiffre le suit. De plus, pour les conversions 'g' et 'G', les zéros finaux ne sont pas retirés du résultat.

L'option ',' signale l'utilisation d'une virgule comme séparateur des milliers. Pour un séparateur conscient de l'environnement linguistique, utilisez plutôt le type de présentation entière 'n'.

Modifié dans la version 3.1 : Ajout de l'option ',' (voir [PEP 378](#)).

L'option '_' demande l'utilisation d'un tiret bas comme séparateur des milliers pour les représentations de nombres flottants et pour les entiers représentés par le type 'd'. Pour les types de représentation d'entiers 'b', 'o', 'x' et 'X', les tirets bas seront insérés tous les 4 chiffres. Pour les autres types de représentation, spécifier cette option est une erreur.

Modifié dans la version 3.6 : Ajout de l'option '_' (voir aussi [PEP 515](#)).

width est un entier en base 10 qui définit la longueur minimale du champ. Si elle n'est pas spécifiée, alors le champ *width* est déterminé par le contenu.

Quand aucun alignement explicite n'est donné, précéder le champs *width* d'un caractère zéro ('0') active le remplissage par zéro des types numériques selon leur signe. Cela est équivalent à un caractère de remplissage *fill* valant '0' avec le type d'alignement *alignment* valant '='.

La valeur *precision* est un nombre en base 10 indiquant combien de chiffres doivent être affichés après la virgule pour une valeur à virgule flottante formatée avec 'f' ou 'F', ou avant et après le point-décimal pour une valeur à virgule flottante formatée avec 'g' ou 'G'. Pour les types non numériques, ce champ indique la taille maximale du champ, autrement dit, combien de caractères du champ sont utilisés. Le spécificateur *precision* n'est pas autorisé sur les entiers.

Finalement, le spécificateur *type* détermine comment la donnée doit être représentée.

Les types disponibles de représentation de chaîne sont :

Type	Signification
's'	Format de chaîne. C'est le type par défaut pour les chaînes de caractères et peut être omis.
None	Pareil que 's'.

Les types disponibles de représentation d'entier sont :

Type	Signification
'b'	Format binaire. Affiche le nombre en base 2.
'c'	Caractère. Convertit l'entier en le caractère Unicode associé avant de l'afficher.
'd'	Entier décimal. Affiche le nombre en base 10.
'o'	Format octal. Affiche le nombre en base 8.
'x'	Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.
'X'	Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9.
'n'	Nombre. Pareil que 'd' si ce n'est que l'environnement linguistique est utilisé afin de déterminer le séparateur de nombres approprié.
None	Pareil que 'd'.

En plus des types de représentation ci-dessus, les entiers peuvent aussi être formatés avec les types de représentation des flottants listés ci-dessous (à l'exception de 'n' et None). Dans ce cas, la fonction `float()` est utilisée pour convertir l'entier en flottant avant le formatage.

Les types de représentation pour les nombres flottants et les valeurs décimales sont :

Type	Signification
'e'	Notation par exposant. Affiche le nombre dans sa notation scientifique en utilisant la lettre “e” pour indiquer l'exposant. La précision par défaut est 6.
'E'	Notation par exposant. Pareil que 'e' sauf l'utilisation de la lettre majuscule “E” comme séparateur.
'f'	Fixed-point notation. Displays the number as a fixed-point number. The default precision is 6.
'F'	Fixed-point notation. Same as 'f', but converts <code>nan</code> to <code>NAN</code> and <code>inf</code> to <code>INF</code> .
'g'	Format général. Pour une précision donnée $p \geq 1$, ceci arrondit le nombre à p chiffres significatifs et puis formate le résultat soit en virgule fixe soit en notation scientifique, en fonction de la magnitude. Les règles précises sont les suivantes : supposons que le résultat formaté avec le type de représentation 'e' et une précision 1 ait un exposant <code>exp</code> . Alors, si $-4 \leq \text{exp} \leq p$, le nombre est formaté avec le type de représentation 'f' et une précision $p-1-\text{exp}$. Sinon, le nombre est formaté avec le type de représentation 'e' et une précision $p-1$. Dans les deux cas, les zéros finaux et non significatifs sont retirés, et la virgule est également retirée s'il n'y a aucun chiffre la suivant. Les valeurs suivantes : infini négatif, infini positif, zéro positif, zéro négatif, <i>not a number</i> sont formatées respectivement par <code>inf</code> , <code>-inf</code> , <code>0</code> , <code>-0</code> et <code>nan</code> , peu importe la précision. Une précision de 0 est interprétée comme une précision de 1. La précision par défaut est 6.
'G'	Format général. Pareil que 'G' si ce n'est que 'E' est utilisé si le nombre est trop grand. Également, la représentation des infinis et de <code>Nan</code> sont en majuscules également.
'n'	Nombre. Pareil que 'g', si ce n'est que l'environnement linguistique est pris en compte pour insérer le séparateur approprié.
'%'	Pourcentage. Multiplie le nombre par 100 et l'affiche en virgule fixe ('f'), suivi d'un symbole pourcent '%'
None	Pareil que 'g', si ce n'est que lorsque la notation en virgule fixe est utilisée, il y a toujours au moins un chiffre derrière la virgule. La précision par défaut celle nécessaire pour afficher la valeur donnée. L'effet visé est de le faire correspondre à la valeur renvoyée par <code>str()</code> altérée par les autres modificateurs de format.

Exemples de formats

Cette section contient des exemples de la syntaxe de `str.format()` et des comparaisons avec l'ancien formatage par `%`.

Dans la plupart des cases, la syntaxe est similaire à l'ancien formatage par `%`, avec l'ajout de `{}` et avec `:` au lieu de `%`. Par exemple : `'%03.2f'` peut être changé en `'{0:3.2f}'`.

The new format syntax also supports new and different options, shown in the following examples.

Accéder à un argument par sa position :

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{} , {} , {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')        # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')    # arguments' indices can be repeated
'abracadabra'
```

Accéder à un argument par son nom :

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.
↳ 81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accéder aux attributs d'un argument :

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.
↳ 0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

Accéder aux éléments d'un argument :

```
>>> coord = (3, 5)
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 3; Y: 5'
```

Remplacer `%s` et `%r` :

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
'repr() shows quotes: \'test1\'; str() doesn\'t: test2'
```


(suite de la page précédente)

```
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
5      5      5      101
6      6      6      110
7      7      7      111
8      8      10     1000
9      9      11     1001
10     A      12     1010
11     B      13     1011
```

6.1.4 Chaînes modèles

Template strings provide simpler string substitutions as described in [PEP 292](#). A primary use case for template strings is for internationalization (i18n) since in that context, the simpler syntax and functionality makes it easier to translate than other built-in string formatting facilities in Python. As an example of a library built on template strings for i18n, see the [flufl.i18n](#) package.

Template strings support $\$$ -based substitutions, using the following rules :

- $\$\$$ est un échappement ; il est remplacé par un simple $\$$.
- $\$identifiant$ dénomme un substituant lié à la clef "identifiant". Par défaut, "identifiant" est restreint à toute chaîne de caractères ASCII alphanumériques sensibles à la casse (avec les *underscores*) commençant avec un *underscore* ou un caractère alphanumérique. Le premier caractère n'étant pas un identifiant après le $\$$ termine la spécification du substituant.
- $\${identifiant}$ est équivalent à $\$identifiant$. Cette notation est requise quand un caractère valide pour une clef de substituant suit directement le substituant mais ne fait pas partie du substituant, comme " $\${noun}ification$ ".

Tout autre présence du symbole $\$$ dans une chaîne résultera en la levée d'une *ValueError*.

Le module *string* fournit une classe *Template* qui implémente ces règles. Les méthodes de *Template* sont :

class *string.Template* (*template*)

Le constructeur prend un seul argument qui est la chaîne du *template*.

substitute (*mapping*, ***kws*)

Applique les substitutions du *template*, et la renvoie dans une nouvelle chaîne. *mapping* est un objet dictionnaire-compatible qui lie les substituants dans le *template*. De même, vous pouvez fournir des arguments mot-clefs tels que les mot-clefs sont les substituants. Quand à la fois *mapping* et *kws* sont donnés et qu'il y a des doublons, les substituants de *kws* sont prioritaires.

safe_substitute (*mapping*, ***kws*)

Comme *substitute()*, si ce n'est qu'au lieu de lever une *KeyError* si un substituant n'est ni dans *mapping*, ni dans *kws*, c'est le nom du substituant inchangé qui apparaît dans la chaîne finale. Également, à l'inverse de *substitute()*, toute autre apparition de $\$$ renverra simplement $\$$ au lieu de lever une exception *ValueError*.

While other exceptions may still occur, this method is called « safe » because it always tries to return a usable string instead of raising an exception. In another sense, *safe_substitute()* may be anything other than

safe, since it will silently ignore malformed templates containing dangling delimiters, unmatched braces, or placeholders that are not valid Python identifiers.

Les instances de la classe `Template` fournissent également un attribut public :

template

C'est l'objet `template` passé comme argument au constructeur. En général, vous ne devriez pas le changer, mais un accès en lecture-seule n'est pas possible à fournir.

Voici un exemple de comment utiliser un `Template` :

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

Usage avancé : vous pouvez faire dériver vos sous-classes de `Template` pour personnaliser la syntaxe des substituants, les caractères délimiteurs, ou l'entière de l'expression rationnelle utilisée pour analyser les chaînes *templates*. Pour faire cela, vous pouvez redéfinir les attributs suivants :

- *delimiter* – La chaîne littérale décrivant le délimiteur pour introduire un substituant. Sa valeur par défaut est `$`. Notez qu'il ne doit *pas* être une expression rationnelle, puisque l'implémentation appellera `re.escape()` sur cette chaîne si nécessaire.
- *idpattern* – L'expression rationnelle décrivant le motif pour les substituants non entourés d'accolades (les accolades sont ajoutées automatiquement si c'est approprié). La valeur par défaut de cette expression rationnelle est `(? -i: [_a-zA-Z] [_a-zA-Z0-9] *)`.

Note : Puisque par défaut *flags* vaut `re.IGNORECASE`, des caractères non-ASCII peuvent correspondre au motif `[a-z]`. C'est pourquoi on utilise une option locale `-i` ici.

Bien que *flags* reste par défaut à `re.IGNORECASE` pour des raisons de compatibilité descendante, vous pouvez le mettre à 0 ou `re.IGNORECASE | re.ASCII` si vous en héritez.

- *flags* – L'indicateur d'expression rationnelle qui sera appliqué lors de la compilation de l'expression rationnelle pour reconnaître les substitutions. La valeur par défaut est `re.IGNORECASE`. Notez que `re.VERBOSE` sera toujours ajouté à l'indicateur. Donc, un *idpattern* personnalisé doit suivre les conventions des expressions rationnelles *verbose*.

Nouveau dans la version 3.2.

Également, vous pouvez fournir le motif d'expression rationnelle en entier en redéfinissant l'attribut *pattern*. Si vous faites cela, la valeur doit être un objet "expression rationnelle" avec quatre groupes de capture de noms. Les groupes de capture correspondent aux règles données au-dessus, ainsi qu'à la règle du substituant invalide :

- *escaped* – Ce groupe lie les séquences échappées (par exemple `$ $`) dans le motif par défaut.
- *named* – Ce groupe lie les substituants non entourés d'accolades ; il ne devrait pas inclure le délimiteur dans le groupe de capture.
- *braced* – Ce groupe lie le nom entouré d'accolades ; il ne devrait inclure ni le délimiteur, ni les accolades dans le groupe de capture.
- *invalid* – Ce groupe lie tout autre motif de délimitation (habituellement, un seul délimiteur) et il devrait apparaître en dernier dans l'expression rationnelle.

6.1.5 Fonctions d'assistance

`string.capwords(s, sep=None)`

Divise l'argument en mots en utilisant `str.split()`, capitalise chaque mot en utilisant `str.capitalize()` et assemble les mots capitalisés en utilisant `str.join()`. Si le second argument optionnel `sep` est absent ou vaut `None`, les séquences de caractères blancs sont remplacées par un seul espace et les espaces débutant et finissant la chaîne sont retirés. Sinon, `sep` est utilisé pour séparer et ré-assembler les mots.

6.2 re — Opérations à base d'expressions rationnelles

Code source : [Lib/re.py](#)

Ce module fournit des opérations sur les expressions rationnelles similaires à celles que l'on trouve dans Perl.

Les motifs, comme les chaînes, à analyser peuvent aussi bien être des chaînes Unicode (`str`) que des chaînes 8-bits (`bytes`). Cependant, les chaînes Unicode et 8-bits ne peuvent pas être mélangées : c'est à dire que vous ne pouvez pas analyser une chaîne Unicode avec un motif 8-bit, et inversement ; de même, lors d'une substitution, la chaîne de remplacement doit être du même type que le motif et la chaîne analysée.

Les expressions rationnelles utilisent le caractère *backslash* (`'\'`) pour indiquer des formes spéciales ou permettre d'utiliser des caractères spéciaux sans en invoquer le sens. Cela entre en conflit avec l'utilisation en Python du même caractère pour la même raison dans les chaînes littérales ; par exemple, pour rechercher un *backslash* littéral il faudrait écrire `'\\'` comme motif, parce que l'expression rationnelle devrait être `\\` et chaque *backslash* doit être représenté par `\\` au sein des chaînes littérales Python.

La solution est d'utiliser la notation des chaînes brutes en Python pour les expressions rationnelles ; Les *backslashes* ne provoquent aucun traitement spécifique dans les chaînes littérales préfixées par `'r'`. Ainsi, `r"\n"` est une chaîne de deux caractères contenant `'\'` et `'n'`, tandis que `"\n"` est une chaîne contenant un unique caractère : un saut de ligne. Généralement, les motifs seront exprimés en Python à l'aide de chaînes brutes.

Il est important de noter que la plupart des opérations sur les expressions rationnelles sont disponibles comme fonctions au niveau du module et comme méthodes des *expressions rationnelles compilées*. Les fonctions sont des raccourcis qui ne vous obligent pas à d'abord compiler un objet *regex*, mais auxquelles manquent certains paramètres de configuration fine.

Voir aussi :

Le module tiers [regex](#), dont l'interface est compatible avec le module `re` de la bibliothèque standard, mais offre des fonctionnalités additionnelles et une meilleure gestion de l'Unicode.

6.2.1 Syntaxe des expressions rationnelles

Une expression rationnelle (*regular expression* ou *RE*) spécifie un ensemble de chaînes de caractères qui lui correspondent ; les fonctions de ce module vous permettent de vérifier si une chaîne particulière correspond à une expression rationnelle donnée (ou si une expression rationnelle donnée correspond à une chaîne particulière, ce qui revient à la même chose).

Les expressions rationnelles peuvent être concaténées pour former de nouvelles expressions : si *A* et *B* sont deux expressions rationnelles, alors *AB* est aussi une expression rationnelle. En général, si une chaîne *p* valide *A* et qu'une autre chaîne *q* valide *B*, la chaîne *pq* validera *AB*. Cela est vrai tant que *A* et *B* ne contiennent pas d'opérations de priorité ; de conditions de frontière entre *A* et *B* ; ou de références vers des groupes numérotés. Ainsi, des expressions complexes peuvent facilement être construites depuis de plus simples expressions primitives comme celles décrites ici. Pour plus de détails sur la théorie et l'implémentation des expressions rationnelles, consultez le livre de Friedl [[Frie09](#)], ou à peu près n'importe quel livre dédié à la construction de compilateurs.

Une brève explication sur le format des expressions rationnelles suit. Pour de plus amples informations et une présentation plus simple, référez-vous au [regex-howto](#).

Les expressions rationnelles peuvent contenir à la fois des caractères spéciaux et ordinaires. Les plus ordinaires, comme 'A', 'a' ou '0' sont les expressions rationnelles les plus simples : elles correspondent simplement à elles-mêmes. Vous pouvez concaténer des caractères ordinaires, ainsi `last` correspond à la chaîne `'last'`. (Dans la suite de cette section, nous écrirons les expressions rationnelles dans ce style spécifique, généralement sans guillemets, et les chaînes à tester 'entourées de simples guillemets'.)

Certains caractères, comme `|` ou `(`, sont spéciaux. Des caractères spéciaux peuvent aussi exister pour les classes de caractères ordinaires, ou affecter comment les expressions rationnelles autour d'eux seront interprétées.

Les caractères de répétition (`*`, `+`, `?`, `{m, n}`, etc.) ne peuvent être directement imbriqués. Cela empêche l'ambiguïté avec le suffixe modificateur non gourmand `?` et avec les autres modificateurs dans d'autres implémentations. Pour appliquer une seconde répétition à une première, des parenthèses peuvent être utilisées. Par exemple, l'expression `(?:a{6})*` valide toutes les chaînes composées d'un nombre de caractères 'a' multiple de six.

Les caractères spéciaux sont :

- (Point.) Dans le mode par défaut, il valide tout caractère à l'exception du saut de ligne. Si l'option `DOTALL` a été spécifiée, il valide tout caractère, saut de ligne compris.
- ^ (Accent circonflexe.) Valide le début d'une chaîne de caractères, ainsi que ce qui suit chaque saut de ligne en mode `MULTILINE`.
- \$ Valide la fin d'une chaîne de caractères, ou juste avant le saut de ligne à la fin de la chaîne, ainsi qu'avant chaque saut de ligne en mode `MULTILINE`. `foo` valide à la fois `foo` et `foobar`, tandis que l'expression rationnelle `foo$` ne correspond qu'à `'foo'`. Plus intéressant, chercher `foo.$` dans `'foo1\nfoo2\n'` trouve normalement `'foo2'`, mais `'foo1'` en mode `MULTILINE`; chercher un simple `$` dans `'foo\n'` trouvera deux correspondances (vides) : une juste avant le saut de ligne, et une à la fin de la chaîne.
- * Fait valider par l'expression rationnelle résultante 0 répétition ou plus de l'expression qui précède, avec autant de répétitions que possible. `ab*` validera "a", "ab" ou "a" suivi de n'importe quel nombre de "b".
- + Fait valider par l'expression rationnelle résultante 1 répétition ou plus de l'expression qui précède. `ab+` validera "a" suivi de n'importe quel nombre non nul de "b"; cela ne validera pas la chaîne "a".
- ? Fait valider par l'expression rationnelle résultante 0 ou 1 répétition de l'expression qui précède. `ab?` correspondra à "a" ou "ab".
- *?, +?, ?? Les qualificateurs `'*'`, `'+'` et `'?'` sont tous *greedy* (gourmands); ils valident autant de texte que possible. Parfois ce comportement n'est pas désiré; si l'expression rationnelle `<.*>` est testée avec la chaîne `'<a>b <c>'`, cela correspondra à la chaîne entière, et non juste à `'<a>'`. Ajouter `?` derrière le qualificateur lui fait réaliser l'opération de façon *non-greedy* (ou *minimal*); le moins de caractères possibles seront validés. Utiliser l'expression rationnelle `<.*?>` validera uniquement `'<a>'`.
- {m} Spécifie qu'exactly *m* copies de l'expression rationnelle qui précède devront être validées; un nombre plus faible de correspondances empêche l'expression entière de correspondre. Par exemple, `a{6}` correspondra exactement à six caractères 'a', mais pas à cinq.
- {m, n} Fait valider par l'expression rationnelle résultante entre *m* et *n* répétitions de l'expression qui précède, cherchant à en valider le plus possible. Par exemple, `a{3, 5}` validera entre 3 et 5 caractères 'a'. Omettre *m* revient à spécifier 0 comme borne inférieure, et omettre *n* à avoir une borne supérieure infinie. Par exemple, `a{4, }b` correspondra à `'aaaab'` ou à un millier de caractères 'a' suivis d'un 'b', mais pas à `'aaab'`. La virgule ne doit pas être omise, auquel cas le modificateur serait confondu avec la forme décrite précédemment.

{m,n}? Fait valider par l'expression rationnelle résultante entre *m* et *n* répétitions de l'expression qui précède, cherchant à en valider le moins possible. Il s'agit de la version non gourmande du précédent qualificateur. Par exemple, dans la chaîne de 6 caractères 'aaaaaa', `a{3,5}` trouvera 5 caractères 'a', alors que `a{3,5}?` n'en trouvera que 3.

**** Échappe les caractères spéciaux (permettant d'identifier des caractères comme '*', '?' et autres) ou signale une séquence spéciale; les séquences spéciales sont décrites ci-dessous.

Si vous n'utilisez pas de chaînes brutes pour exprimer le motif, souvenez-vous que Python utilise aussi le *backslash* comme une séquence d'échappement dans les chaînes littérales; si la séquence d'échappement n'est pas reconnue par l'interpréteur Python, le *backslash* et les caractères qui le suivent sont inclus dans la chaîne renvoyée. Cependant, si Python reconnaît la séquence, le *backslash* doit être doublé (pour ne plus être reconnu). C'est assez compliqué et difficile à comprendre, c'est pourquoi il est hautement recommandé d'utiliser des chaînes brutes pour tout sauf les expressions les plus simples.

[] Utilisé pour indiquer un ensemble de caractères. Dans un ensemble :

- Les caractères peuvent être listés individuellement, e.g. `[amk]` correspondra à 'a', 'm' ou 'k'.
- Des intervalles de caractères peuvent être indiqués en donnant deux caractères et les séparant par un '-', par exemple `[a-z]` correspondra à toute lettre minuscule *ASCII*, `[0-5][0-9]` à tous nombres de deux chiffres entre 00 et 59, et `[0-9A-Fa-f]` correspondra à n'importe quel chiffre hexadécimal. Si '-' est échappé (`[a\ -z]`) ou s'il est placé comme premier ou dernier caractère (e.g. `[-a]` ou `[a-]`), il correspondra à un '-' littéral.
- Les caractères spéciaux perdent leur sens à l'intérieur des ensembles. Par exemple, `[+*]` validera chacun des caractères littéraux '+', '*', ou ' '.
- Les classes de caractères telles que `\w` ou `\S` (définies ci-dessous) sont aussi acceptées à l'intérieur d'un ensemble, bien que les caractères correspondant dépendent de quel mode est actif entre *ASCII* et *LOCALE*.
- Les caractères qui ne sont pas dans un intervalle peuvent être trouvés avec l'ensemble complémentaire (*complementing*). Si le premier caractère de l'ensemble est '^', tous les caractères qui *ne sont pas* dans l'ensemble seront validés. Par exemple, `^[5]` correspondra à tout caractère autre que '5' et `^[^]` validera n'importe quel caractère excepté '^'. '^' n'a pas de sens particulier s'il n'est pas le premier caractère de l'ensemble.
- Pour insérer un ']' littéral dans un ensemble, il faut le précéder d'un *backslash* ou le placer au début de l'ensemble. Par exemple, `[() \[\] {}]` et `[] () [\{\}]` vont tous deux correspondre à une parenthèse, un crochet ou une accolade.

| *A|B*, où *A* et *B* peuvent être deux expressions rationnelles arbitraires, crée une expression rationnelle qui validera soit *A* soit *B*. Un nombre arbitraire d'expressions peuvent être séparées de cette façon par des '|'. Cela peut aussi être utilisé au sein de groupes (voir ci-dessous). Quand une chaîne cible est analysée, les expressions séparées par '|' sont essayées de la gauche vers la droite. Quand un motif correspond complètement, cette branche est acceptée. Cela signifie qu'une fois que *A* correspond, *B* ne sera pas testée plus loin, même si elle pourrait provoquer une plus ample correspondance. En d'autres termes, l'opérateur '|' n'est jamais gourmand. Pour valider un '|' littéral, utilisez `\|`, ou enveloppez-le dans une classe de caractères, comme `[|]`.

(...) Valide n'importe quelle expression rationnelle comprise entre les parenthèses, et indique le début et la fin d'un groupe; le contenu d'un groupe peut être récupéré après qu'une analyse a été effectuée et peut être réutilisé plus loin dans la chaîne avec une séquence spéciale `\number`, décrite ci-dessous. Pour écrire des '(' ou ')' littéraux, utilisez `\(` (ou `\)`), ou enveloppez-les dans une classe de caractères : `[()]`.

(?...) Il s'agit d'une notation pour les extensions (un '?' suivant une '(' n'a pas de sens autrement). Le premier caractère après le '?' détermine quel sens donner à l'expression. Les extensions ne créent généralement pas de nouveaux groupes; `(?P<name>...)` est la seule exception à la règle. Retrouvez ci-dessous la liste des extensions actuellement supportées.

(?aiLmsux) (Une lettre ou plus de l'ensemble 'a', 'i', 'L', 'm', 's', 'u', 'x'.) Le groupe valide la chaîne vide ; les lettres activent les modes correspondant : *re.A* (validation ASCII seulement), *re.I* (ignorer la casse), *re.L* (dépendant de la locale), *re.M* (multi-ligne), *re.S* (les points correspondent à tous les caractères), *re.U* (support d'Unicode) et *re.X* (verbeux), pour l'ensemble de l'expression rationnelle. (Les options dans décrites dans la section *Contenu du module*.) C'est utile si vous souhaitez préciser l'option dans l'expression rationnelle, plutôt qu'en passant un argument *flag* à la fonction *re.compile()*. Les options devraient être spécifiées en premier dans la chaîne de l'expression.

(?:...) Une version sans capture des parenthèses habituelles. Valide n'importe quelle expression rationnelle à l'intérieur des parenthèses, mais la sous-chaîne correspondant au groupe *ne peut pas* être récupérée après l'analyse ou être référencée plus loin dans le motif.

(?imsx-imsx:...) (Zéro lettres ou plus de l'ensemble 'i', 'm', 's', 'x', optionnellement suivies par '-' ainsi qu'une ou plusieurs lettres du même ensemble.) Les lettres activent ou désactivent les options correspondantes : *re.I* (ignorer la casse), *re.M* (multi-ligne), *re.S* (les points correspondent à tous les caractères) et *re.X* (verbeux), pour cette partie de l'expression. (Les options sont décrites dans la section *Contenu du module*.) Nouveau dans la version 3.6.

(?P<name>...) Similaires aux parenthèses habituelles, mais la sous-chaîne validée par le groupe est accessible via le nom *name* du groupe symbolique. Les noms de groupes doivent être des identifiants Python valides, et chaque nom de groupe ne doit être défini qu'une seule fois dans une expression rationnelle. Un groupe symbolique est aussi un groupe numéroté, de la même manière que si le groupe n'était pas nommé.

Les groupes nommés peuvent être référencés dans trois contextes. Si le motif est `(?P<quote>["])*?(?P=quote)` (i.e. correspondant à une chaîne entourée de guillemets simples ou doubles) :

Contexte de référence au groupe <i>quote</i>	Manières de le référencer
lui-même dans le même motif	<ul style="list-style-type: none"> — <code>(?P=quote)</code> (comme vu) — <code>\1</code>
en analysant l'objet résultat <i>m</i>	<ul style="list-style-type: none"> — <code>m.group('quote')</code> — <code>m.end('quote')</code> (etc.)
dans une chaîne passée à l'argument <i>repl</i> de <i>re.sub()</i>	<ul style="list-style-type: none"> — <code>\g<quote></code> — <code>\g<1></code> — <code>\1</code>

(?P=name) Une référence arrière à un groupe nommé ; elle correspond à n'importe quel texte validé plus tôt par le groupe nommé *name*.

(?#...) Un commentaire ; le contenu des parenthèses est simplement ignoré.

(?=...) Valide si ... valide la suite, mais ne consomme rien de la chaîne. On appelle cela une assertion *lookahead*. Par exemple, `Isaac (?=Asimov)` correspondra à la chaîne 'Isaac' `` seulement si elle est suivie par `` 'Asimov'.

(?!...) Valide si ... ne valide pas la suite. C'est une assertion *negative lookahead*. Par exemple, `Isaac (?!Asimov)` correspondra à la chaîne 'Isaac ' seulement si elle *n'est pas* suivie par 'Asimov'.

(?<=...) Valide si la position courante dans la chaîne est précédée par une correspondance sur ... qui se termine à la position courante. On appelle cela une *positive lookbehind assertion*. (?<=abc)def cherchera une correspondance dans 'abcdef', puisque le *lookbehind** mettra de côté 3 caractères et vérifiera que le motif contenu correspond. Le motif ne devra correspondre qu'à des chaînes de taille fixe, cela veut dire que abc ou a|b` sont autorisées, mais pas `a* ou a{3,4}. Notez que les motifs qui commencent par des assertions *lookbehind* positives ne peuvent pas correspondre au début de la chaîne analysée ; vous préférerez sûrement utiliser la fonction `search()` plutôt que la fonction `match()` :

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

Cet exemple recherche un mot suivi d'un trait d'union :

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

Modifié dans la version 3.5 : Ajout du support des références aux groupes de taille fixe.

(?!...) Valide si la position courante dans la chaîne n'est pas précédée par une correspondance sur ... On appelle cela une *negative lookbehind assertion*. À la manière des assertions *lookbehind* positives, le motif contenu ne peut que correspondre à des chaînes de taille fixe. Les motifs débutant par une assertion *lookbehind* négative peuvent correspondre au début de la chaîne analysée.

(?(id/name)yes-pattern|no-pattern) Essaiera de faire la correspondance avec yes-pattern si le groupe indiqué par *id* ou *name* existe, et avec no-pattern s'il n'existe pas. no-pattern est optionnel et peut être omis. Par exemple, (<)?(\w+@\w+(?:\.\w+)+)(?(1)>|)\$ est un motif simpliste pour identifier une adresse courriel, qui validera '<user@host.com>' ainsi que 'user@host.com' mais pas '<user@host.com' ni 'user@host.com>'.

Les séquences spéciales sont composées de '\ ' et d'un caractère de la liste qui suit. Si le caractère ordinaire n'est pas un chiffre *ASCII* ou une lettre *ASCII*, alors l'expression rationnelle résultante validera le second caractère de la séquence. Par exemple, \\$ correspond au caractère '\$'.

\number Correspond au contenu du groupe du même nombre. Les groupes sont numérotés à partir de 1. Par exemple, (.+) \1 correspond à 'the the' ou '55 55', mais pas à 'thethe' (notez l'espace après le groupe). Cette séquence spéciale ne peut être utilisée que pour faire référence aux 99 premiers groupes. Si le premier chiffre de *number* est 0, ou si *number* est un nombre octal de 3 chiffres, il ne sera pas interprété comme une référence à un groupe, mais comme le caractère à la valeur octale *number*. À l'intérieur des '[' et ']' d'une classe de caractères, tous les échappements numériques sont traités comme des caractères.

\A Correspond uniquement au début d'une chaîne de caractères.

\b Correspond à la chaîne vide, mais uniquement au début ou à la fin d'un mot. Un mot est défini comme une séquence de « caractères de mots ». Notez que formellement, \b est défini comme la liaison entre \w et \W (et inversement), ou entre \w et le début/fin d'un mot. Cela signifie que r'\bfoo\b' validera 'foo', 'foo.', '(foo)' ou 'bar foo baz' mais pas 'foobar' ou 'foo3'.

Les caractères alphanumériques Unicode sont utilisés par défaut dans les motifs Unicode, mais cela peut être changé en utilisant l'option *ASCII*. Les délimitations de mots sont déterminées par la locale si l'option *LOCALE* est utilisée. À l'intérieur d'un intervalle de caractères, \b représente le caractère *backspace*, par compatibilité avec les chaînes littérales Python.

\B Correspond à la chaîne vide, mais uniquement quand elle *n'est pas* au début ou à la fin d'un mot. Cela signifie que `r'py\B'` valide `'python'`, `'py3'` ou `'py2'`, mais pas `'py'`, `'py.'` ou `'py!'`. `\B` est simplement l'opposé de `\b`, donc les caractères de mots dans les motifs Unicode sont les alphanumériques et tirets bas Unicode, bien que cela puisse être changé avec l'option `ASCII`. Les délimitations de mots sont déterminées par la locale si l'option `LOCALE` est utilisée.

\d

Pour les motifs Unicode (`str`) : Valide n'importe quel chiffre décimal Unicode (soit tout caractère Unicode de catégorie `[Nd]`). Cela inclue `[0-9]`, mais aussi bien d'autres caractères de chiffres. Si l'option `ASCII` est utilisée, seuls les caractères de la classe `[0-9]` correspondront (mais l'option affectant l'expression rationnelle entière, il peut être préférable dans ce genre de cas d'utiliser un `[0-9]` explicite).

Pour les motifs 8-bit (bytes) : Valide n'importe quel chiffre décimal; équivalent à `[0-9]`.

\D Valide tout caractère qui n'est pas un chiffre décimal. C'est l'opposé de `\d`. Si l'option `ASCII` est utilisée, cela devient équivalent à `[^0-9]` (mais l'option affectant l'expression rationnelle entière, il peut être préférable dans ce genre de cas d'utiliser explicitement `[^0-9]`).

\s

Pour les motifs Unicode (`str`) : Valide les caractères d'espace Unicode (qui incluent `[\t\n\r\f\v]` et bien d'autres, comme les espaces insécables requises par les règles typographiques de beaucoup de langues). Si l'option `ASCII` est utilisée, seuls les caractères de la classe `[\t\n\r\f\v]` sont validés (mais l'option affectant l'expression rationnelle entière, il peut être préférable dans ce genre de cas d'utiliser un `[\t\n\r\f\v]` explicite).

Pour les motifs 8-bit (bytes) : Valide les caractères considérés comme des espaces dans la table ASCII; équivalent à `[\t\n\r\f\v]`.

\S Valide tout caractère qui n'est pas un caractère d'espace. c'est l'opposé de `\s`. Si l'option `ASCII` est utilisée, cela devient équivalent à `[^\t\n\r\f\v]` (mais l'option affectant l'expression rationnelle entière, il peut être préférable dans ce genre de cas d'utiliser un `[^\t\n\r\f\v]` explicite).

\w

Pour les motifs Unicode (`str`) : Valide les caractères Unicode de mot; cela inclut la plupart des caractères qui peuvent être compris dans un mot d'une quelconque langue, aussi bien que les nombres et les tirets bas. Si l'option `ASCII` est utilisée, seuls les caractères de la classe `[a-zA-Z0-9_]` sont validés (mais l'option affectant l'expression rationnelle entière, il peut être préférable dans ce genre de cas d'utiliser un `[a-zA-Z0-9_]` explicite).

Pour les motifs 8-bit (bytes) : Valide les caractères alphanumériques de la table ASCII; équivalent à `[a-zA-Z0-9_]`. Si l'option `LOCALE` est utilisée, les caractères considérés alphanumériques dans la locale et le tiret bas seront acceptés.

\W Valide tout caractère qui n'est pas un caractère de mot. C'est l'opposé de `\w`. Si l'option `ASCII` est utilisée, cela devient équivalent à `[^a-zA-Z0-9_]` (mais l'option affectant l'expression rationnelle entière, il peut être préférable dans ce genre de cas d'utiliser un `[^a-zA-Z0-9_]` explicite). Si l'option `LOCALE` est utilisée, les caractères considérés alphanumériques dans la locale courante, et le tiret bas, seront acceptés.

\Z Correspond uniquement à la fin d'une chaîne de caractères.

La plupart des échappements standards supportés par les chaînes littérales sont aussi acceptés par l'analyseur d'expressions rationnelles :

<code>\a</code>	<code>\b</code>	<code>\f</code>	<code>\n</code>
<code>\r</code>	<code>\t</code>	<code>\u</code>	<code>\U</code>
<code>\v</code>	<code>\x</code>	<code>\\</code>	

(Notez que `\b` est utilisé pour représenter les bornes d'un mot, et signifie « *backspace* » uniquement à l'intérieur d'une classe de caractères.)

Les séquences d'échappement `'\u'` et `'\U'` sont seulement reconnues dans les motifs Unicode. Dans les motifs de *byte*, ce sont des erreurs.

Les séquences octales d'échappement sont incluses dans une forme limitée. Si le premier chiffre est un 0, ou s'il y a trois chiffres octaux, la séquence est considérée comme octale. Autrement, il s'agit d'une référence vers un groupe. Comme pour les chaînes littérales, les séquences octales ne font jamais plus de 3 caractères de long.

Modifié dans la version 3.3 : Les séquences d'échappement `'\u'` et `'\U'` ont été ajoutées.

Modifié dans la version 3.6 : Les séquences inconnues composées de `'\'` et d'une lettre ASCII sont maintenant des erreurs.

6.2.2 Contenu du module

Le module définit plusieurs fonctions, constantes, et une exception. Certaines fonctions sont des versions simplifiées des méthodes plus complètes des expressions rationnelles compilées. La plupart des applications non triviales utilisent toujours la version compilée.

Modifié dans la version 3.6 : Les constantes d'options sont maintenant des instances de `RegexFlag`, sous-classe de `enum.IntFlag`.

`re.compile(pattern, flags=0)`

Compile un motif vers une *expression rationnelle* compilée, dont les méthodes `match()` et `search()`, décrites ci-dessous, peuvent être utilisées pour analyser des textes.

Le comportement des expressions peut être modifié en spécifiant une valeur *flags*. Les valeurs sont comprises dans les variables suivantes, et peuvent être combinées avec un *ou* bit-à-bit (opérateur `|`).

La séquence

```
prog = re.compile(pattern)
result = prog.match(string)
```

est équivalente à

```
result = re.match(pattern, string)
```

mais utiliser `re.compile()` et sauvegarder l'expression rationnelle renvoyée pour la réutiliser est plus efficace quand l'expression est amenée à être utilisée plusieurs fois dans un même programme.

Note : Les versions compilées des motifs les plus récents passés à `re.compile()` et autres fonctions d'analyse du module sont mises en cache, ainsi les programmes qui n'utilisent que quelques expressions rationnelles en même temps n'ont pas à s'inquiéter de la compilation de ces expressions.

`re.A`

`re.ASCII`

Fait correspondre à `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` et `\S` des caractères ASCII seulement, plutôt qu'Unicode. Cela n'a du sens que pour les motifs Unicode, et est ignoré pour les motifs 8-bit. Correspond à l'option de groupe `(?a)`.

Notez que par compatibilité envers les versions précédentes, l'option `re.U` existe toujours (ainsi que son synonyme `re.UNICODE` et sa version embarquée `(?u)`), mais elles sont redondantes en Python 3 depuis que l'analyse est

faite en Unicode par défaut pour les chaînes de caractères (et que l'analyse Unicode n'est pas permise pour les chaînes 8-bit).

re.DEBUG

Affiche des informations de débogage à propos de l'expression compilée. N'a pas d'option de groupe équivalente.

re.I

re.IGNORECASE

Effectue une analyse indépendante de la casse. Les motifs tels que `[A-Z]` accepteront donc les caractères minuscules. L'analyse Unicode complète (tel que `Ü` correspondant à `ü`) fonctionne aussi, tant que l'option `re.ASCII` n'est pas utilisée. La locale n'affecte pas cette option, tant que l'option `re.LOCALE` n'est pas utilisée. Correspond au marqueur de groupe `(?i)`.

Note that when the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the `IGNORECASE` flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters : “İ” (U+0130, Latin capital letter I with dot above), “ı” (U+0131, Latin small letter dotless i), “ſ” (U+017F, Latin small letter long s) and “K” (U+212A, Kelvin sign). If the `ASCII` flag is used, only letters “a” to “z” and “A” to “Z” are matched (but the flag affects the entire regular expression, so in such cases using an explicit `(?-i:[a-zA-Z])` may be a better choice).

re.L

re.LOCALE

Fait dépendre de la locale courante : `\w`, `\W`, `\b`, `\B`, et l'analyse insensible à la casse. Cette option peut être utilisée avec les motifs en *bytes*. L'utilisation de cette option est déconseillée à cause du mécanisme de locale très peu fiable, et ne gérant qu'une « culture » à la fois, et ne fonctionnant que pour les locales 8-bits. L'analyse Unicode est déjà activée par défaut dans Python 3 pour les motifs Unicode (*str*), et elle est capable de gérer plusieurs locales et langages. Correspond à l'option de groupe `(?L)`.

Modifié dans la version 3.6 : `re.LOCALE` ne peut être utilisée qu'avec les motifs 8-bit et n'est pas compatible avec `re.ASCII`.

re.M

re.MULTILINE

Quand spécifiée, le caractère `^` correspond au début d'une chaîne et au début d'une ligne (caractère suivant directement le saut de ligne) ; et le caractère `$` correspond à la fin d'une chaîne et à la fin d'une ligne (juste avant le saut de ligne). Par défaut, `^` correspond uniquement au début de la chaîne, et `$` uniquement à la fin de la chaîne, ou immédiatement avant le saut de ligne (s'il y a) à la fin de la chaîne. Correspond à l'option de groupe `(?m)`.

re.S

re.DOTALL

Fait correspondre tous les caractères possibles à `.`, incluant le saut de ligne ; sans cette option, `.` correspondrait à tout caractère à l'exception du saut de ligne. Correspond à l'option de groupe `(?s)`.

re.X

re.VERBOSE

Cette option vous autorise à écrire des expressions rationnelles qui présentent mieux et sont plus lisibles en vous permettant de séparer visuellement les sections logiques du motif et d'ajouter des commentaires. Les caractères d'espacement à l'intérieur du motif sont ignorés, sauf à l'intérieur des classes de caractères ou quand précédés d'un *backslash* non échappé, ou dans des séquences comme `*?`, `(?:` or `(?P<...>`. Quand une ligne contient un `#` qui n'est pas dans une classe de caractères ou précédé d'un *backslash* non échappé, tous les caractères depuis le `#` le plus à gauche jusqu'à la fin de la ligne sont ignorés.

Cela signifie que les deux expressions rationnelles suivantes qui valident un nombre décimal sont fonctionnellement égales :

```
a = re.compile(r"""
\d +   # the integral part
\.     # the decimal point
\d *   # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

Correspond à l'option de groupe `(?x)`.

`re.search(pattern, string, flags=0)`

Analyse *string* à la recherche du premier emplacement où l'expression rationnelle *pattern* trouve une correspondance, et renvoie l'*objet de correspondance* trouvé. Renvoie `None` si aucune position dans la chaîne ne valide le motif ; notez que cela est différent de trouver une correspondance avec une chaîne vide à un certain endroit de la chaîne.

`re.match(pattern, string, flags=0)`

Si zéro ou plus caractères au début de *string* correspondent à l'expression rationnelle *pattern*, renvoie l'*objet de correspondance* trouvé. Renvoie `None` si la chaîne ne correspond pas au motif ; notez que cela est différent d'une correspondance avec une chaîne vide.

Notez que même en mode *MULTILINE*, `re.match()` ne validera qu'au début de la chaîne et non au début de chaque ligne.

Si vous voulez trouver une correspondance n'importe où dans *string*, utilisez plutôt `search()` (voir aussi `search()` vs. `match()`).

`re.fullmatch(pattern, string, flags=0)`

Si l'entièreté de la chaîne *string* correspond à l'expression rationnelle *pattern*, renvoie l'*objet de correspondance* trouvé. Renvoie `None` si la chaîne ne correspond pas au motif ; notez que cela est différent d'une correspondance avec une chaîne vide.

Nouveau dans la version 3.4.

`re.split(pattern, string, maxsplit=0, flags=0)`

Sépare *string* selon les occurrences de *pattern*. Si des parenthèses de capture sont utilisées dans *pattern*, alors les textes des groupes du motif sont aussi renvoyés comme éléments de la liste résultante. Si *maxsplit* est différent de zéro, il ne pourra y avoir plus de *maxsplit* séparations, et le reste de la chaîne sera renvoyé comme le dernier élément de la liste.

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ',', 'words', ',', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

S'il y a des groupes de capture dans le séparateur et qu'ils trouvent une correspondance au début de la chaîne, le résultat commencera par une chaîne vide. La même chose se produit pour la fin de la chaîne :

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ',', 'words', '...', '']
```

De cette manière, les séparateurs sont toujours trouvés aux mêmes indices relatifs dans la liste résultante.

Note : `split()` ne sépare actuellement pas une chaîne sur une correspondance vide. Par exemple :

```
>>> re.split('x*', 'axbc')
['a', 'bc']
```

Même si `'x*'` correspond aussi à 0 "x" avant "a", entre "b" et "c", et après "c", ces correspondances sont actuellement ignorées. Le comportement correct (i.e. découper aussi sur les correspondances vides et renvoyer `['', 'a', 'b', 'c', '']`) sera implémenté dans les futures versions de Python, mais comme cela constitue un changement incompatible avec les précédentes, une *FutureWarning* sera levée pendant la transition.

Les motifs qui ne peuvent correspondre qu'à des chaînes vides ne permettent actuellement pas de découper la chaîne. Puisque cela ne correspond pas au comportement voulu, une *ValueError* sera levée à partir de Python 3.5 :

```
>>> re.split("^$", "foo\n\nbar\n", flags=re.M)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    ...
ValueError: split() requires a non-empty pattern match.
```

Modifié dans la version 3.1 : Ajout de l'argument optionnel *flags*.

Modifié dans la version 3.5 : Découper sur un motif qui peut correspondre à une chaîne vide lève maintenant un avertissement. Les motifs qui ne peuvent correspondre qu'à des chaînes vides sont maintenant rejetés.

re.findall (*pattern*, *string*, *flags*=0)

Renvoie toutes les correspondances de *pattern* dans *string* qui ne se chevauchent pas, sous forme d'une liste de chaînes. Le chaîne *string* est analysée de la gauche vers la droite, et les correspondances sont renvoyées dans l'ordre où elles sont trouvées. Si un groupe ou plus sont présents dans le motif, renvoie une liste de groupes ; il s'agira d'une liste de *tuples* si le motif a plus d'un groupe. Les correspondances vides sont incluses dans le résultat.

Note : Due to the limitation of the current implementation the character following an empty match is not included in a next match, so `findall(r'^|\w+', 'two words')` returns `['', 'wo', 'words']` (note missed « t »). This is changed in Python 3.7.

re.finditer (*pattern*, *string*, *flags*=0)

Renvoie un *iterator* produisant des *objets de correspondance* pour toutes les correspondances non chevauchantes de l'expression rationnelle *pattern* sur la chaîne *string*. *string* est analysée de la gauche vers la droite, et les correspondances sont renvoyées dans l'ordre où elles sont trouvées. Les correspondances vides sont incluses dans le résultat. Consultez la note à propos de `findall()`.

re.sub (*pattern*, *repl*, *string*, *count*=0, *flags*=0)

Renvoie la chaîne obtenue en remplaçant les occurrences (sans chevauchement) les plus à gauche de *pattern* dans *string* par le remplacement *repl*. Si le motif n'est pas trouvé, *string* est renvoyée inchangée. *repl* peut être une chaîne de caractères ou une fonction ; si c'est une chaîne, toutes les séquences d'échappement qu'elle contient sont traduites. Ainsi, `\n` est convertie en un simple saut de ligne, `\r` en un retour chariot, et ainsi de suite. Les séquences inconnues telles que `&` sont laissées intactes. Les références arrières, telles que `\6`, sont remplacées par la sous-chaîne correspondant au groupe 6 dans le motif. Par exemple :

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*(\s*\n):',
...       r'static PyObject*\npy_\1(void)\n{',
...       'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

Si *repl* est une fonction, elle est appelée pour chaque occurrence non chevauchante de *pattern*. La fonction prend comme argument un *objet de correspondance*, et renvoie la chaîne de remplacement. Par exemple :

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro---gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

Le motif peut être une chaîne de caractères ou un *objet expression rationnelle*.

L'argument optionnel *count* est le nombre maximum d'occurrences du motif à remplacer : *count* ne doit pas être un nombre négatif. Si omis ou nul, toutes les occurrences seront remplacées. Les correspondances vides avec le motif sont remplacées uniquement quand elles ne sont pas adjacentes à une précédente correspondance, ainsi `sub('x*', '-', 'abc')` renvoie `'-a-b-c-'`.

Dans les arguments *repl* de type *string*, en plus des séquences d'échappement et références arrières décrites au-dessus, `\g<name>` utilisera la sous-chaîne correspondant au groupe nommé *name*, comme défini par la syntaxe `(?P<name>...)`. `\g<number>` utilise le groupe numéroté associé; `\g<2>` est ainsi équivalent à `\2`, mais n'est pas ambigu dans un remplacement tel que `\g<2>0`, `\20` serait interprété comme une référence au groupe 20, et non une référence au groupe 2 suivie par un caractère littéral '0'. La référence arrière `\g<0>` est remplacée par la sous-chaîne entière validée par l'expression rationnelle.

Modifié dans la version 3.1 : Ajout de l'argument optionnel *flags*.

Modifié dans la version 3.5 : Les groupes sans correspondance sont remplacés par une chaîne vide.

Modifié dans la version 3.6 : Les séquences d'échappement inconnues dans *pattern* formées par `'\'` et une lettre ASCII sont maintenant des erreurs.

Deprecated since version 3.5, will be removed in version 3.7 : Les séquences d'échappement dans *repl* formées d'un `'\'` et d'une lettre ASCII lèvent maintenant un avertissement de dépréciation et seront interdites en Python 3.7.

re.subn (*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Réalise la même opération que *sub()*, mais renvoie un *tuple* (nouvelle_chaîne, nombre_de_substitutions_réalisées).

Modifié dans la version 3.1 : Ajout de l'argument optionnel *flags*.

Modifié dans la version 3.5 : Les groupes sans correspondance sont remplacés par une chaîne vide.

re.escape (*pattern*)

Échappe tous les caractères de *pattern* à l'exception des lettres ASCII, des nombres et de `'_'`. Cela est utile si vous voulez valider une quelconque chaîne littérale qui pourrait contenir des métacaractères d'expressions rationnelles. Par exemple :

```
>>> print(re.escape('python.exe'))
python\.exe

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789\!\#$%&'*\+,\-.\^_\`|\~\.:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print(''.join(map(re.escape, sorted(operators, reverse=True))))
\|\/\-\|\/\+|\*\*|\/\*\|
```

Ces fonctions ne doivent pas être utilisées pour la chaîne de remplacement dans *sub()* et *subn()*, seuls les antislash devraient être échappés. Par exemple :

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\d', r'\d'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

Modifié dans la version 3.3 : Le caractère `'_'` n'est plus échappé.

re.purge ()

Vide le cache d'expressions rationnelles.

exception re.error (*msg*, *pattern=None*, *pos=None*)

Exception levée quand une chaîne passée à l'une des fonctions ici présentes n'est pas une expression rationnelle valide (contenant par exemple une parenthèse non fermée) ou quand d'autres erreurs se produisent durant la compilation ou l'analyse. Il ne se produit jamais d'erreur si une chaîne ne contient aucune correspondance pour un motif. Les instances de l'erreur ont les attributs additionnels suivants :

msg

Le message d'erreur non formaté.

pattern

Le motif d'expression rationnelle.

pos

L'index dans *pattern* où la compilation a échoué (peut valoir None).

lineno

La ligne correspondant à *pos* (peut valoir None).

colno

La colonne correspondant à *pos* (peut valoir None).

Modifié dans la version 3.5 : Ajout des attributs additionnels.

6.2.3 Objets d'expressions rationnelles

Les expressions rationnelles compilées supportent les méthodes et attributs suivants :

`regex.search(string[, pos[, endpos]])`

Analyse *string* à la recherche du premier emplacement où l'expression rationnelle trouve une correspondance, et envoie l'*objet de correspondance* trouvé. Renvoie None si aucune position dans la chaîne ne satisfait le motif ; notez que cela est différent que de trouver une correspondance vide dans la chaîne.

Le second paramètre *pos* (optionnel) donne l'index dans la chaîne où la recherche doit débiter ; il vaut 0 par défaut. Cela n'est pas complètement équivalent à un *slicing* sur la chaîne ; le caractère de motif '[^]' correspond au début réel de la chaîne et aux positions juste après un saut de ligne, mais pas nécessairement à l'index où la recherche commence.

Le paramètre optionnel *endpos* limite la longueur sur laquelle la chaîne sera analysée ; ce sera comme si la chaîne faisait *endpos* caractères de long, donc uniquement les caractères de *pos* à *endpos* - 1 seront analysés pour trouver une correspondance. Si *endpos* est inférieur à *pos*, aucune correspondance ne sera trouvée ; dit autrement, avec *rx* une expression rationnelle compilée, `rx.search(string, 0, 50)` est équivalent à `rx.search(string[:50], 0)`.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")      # Match at index 0
<_sre.SRE_Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)   # No match; search doesn't include the "d"
```

`regex.match(string[, pos[, endpos]])`

Si zéro caractère ou plus au début de *string* correspondent à cette expression rationnelle, renvoie l'*objet de correspondance* trouvé. Renvoie None si la chaîne ne correspond pas au motif ; notez que cela est différent d'une correspondance vide.

Les paramètres optionnels *pos* et *endpos* ont le même sens que pour la méthode `search()`.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")       # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)    # Match as "o" is the 2nd character of "dog".
<_sre.SRE_Match object; span=(1, 2), match='o'>
```

Si vous voulez une recherche n'importe où dans *string*, utilisez plutôt `search()` (voir aussi `search()` vs. `match()`).

`regex.fullmatch(string[, pos[, endpos]])`

Si la chaîne *string* entière valide l'expression rationnelle, renvoie l'*objet de correspondance* associé. Renvoie None si la chaîne ne correspond pas au motif ; notez que cela est différent d'une correspondance vide.

Les paramètres optionnels *pos* et *endpos* ont le même sens que pour la méthode `search()`.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")   # No match as "o" is not at the start of "dog".
>>> pattern.fullmatch("ogre")  # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3) # Matches within given limits.
<_sre.SRE_Match object; span=(1, 3), match='og'>
```

Nouveau dans la version 3.4.

`regex.split (string, maxsplit=0)`

Identique à la fonction `split()`, en utilisant le motif compilé.

`regex.findall (string[, pos[, endpos]])`

Similaire à la fonction `findall()`, en utilisant le motif compilé, mais accepte aussi des paramètres `pos` et `endpos` optionnels qui limitent la région de recherche comme pour `search()`.

`regex.finditer (string[, pos[, endpos]])`

Similaire à la fonction `finditer()`, en utilisant le motif compilé, mais accepte aussi des paramètres `pos` et `endpos` optionnels qui limitent la région de recherche comme pour `search()`.

`regex.sub (repl, string, count=0)`

Identique à la fonction `sub()`, en utilisant le motif compilé.

`regex.subn (repl, string, count=0)`

Identique à la fonction `subn()`, en utilisant le motif compilé.

`regex.flags`

Les options de validation de l'expression rationnelle. Il s'agit d'une combinaison des options données à `compile()`, des potentielles options `(? . . .)` dans le motif, et des options implicites comme `UNICODE` si le motif est une chaîne Unicode.

`regex.groups`

Le nombre de groupes de capture dans le motif.

`regex.groupindex`

Un dictionnaire associant les noms de groupes symboliques définis par `(?P<id>)` aux groupes numérotés. Le dictionnaire est vide si aucun groupe symbolique n'est utilisé dans le motif.

`regex.pattern`

La chaîne de motif depuis laquelle l'objet expression rationnelle a été compilé.

6.2.4 Objets de correspondance

Les objets de correspondance ont toujours une valeur booléenne `True`. Puisque `match()` et `search()` renvoient `None` quand il n'y a pas de correspondance, vous pouvez tester s'il y a eu correspondance avec une simple instruction `if` :

```
match = re.search(pattern, string)
if match:
    process(match)
```

Les objets de correspondance supportent les méthodes et attributs suivants :

`match.expand (template)`

Renvoie la chaîne obtenue en substituant les séquences d'échappement du gabarit `template`, comme réalisé par la méthode `sub()`. Les séquences comme `\n` sont converties vers les caractères appropriés, et les références arrières numériques (`\1`, `\2`) et nommées (`\g<1>`, `\g<name>`) sont remplacées par les contenus des groupes correspondant.

Modifié dans la version 3.5 : Les groupes sans correspondance sont remplacés par une chaîne vide.

`match.group ([group1, ...])`

Renvoie un ou plus sous-groupes de la correspondance. Si un seul argument est donné, le résultat est une chaîne simple ; s'il y a plusieurs arguments, le résultat est un *tuple* comprenant un élément par argument. Sans arguments, `group1` vaut par défaut zéro (la correspondance entière est renvoyée). Si un argument `groupN` vaut zéro, l'élément associé sera la chaîne de correspondance entière ; s'il est dans l'intervalle fermé `[1..99]`, c'est la correspondance avec le groupe de parenthèses associé. Si un numéro de groupe est négatif ou supérieur au nombre de groupes définis dans le motif, une exception `IndexError` est levée. Si un groupe est contenu dans une partie du motif

qui n'a aucune correspondance, l'élément associé sera `None`. Si un groupe est contenu dans une partie du motif qui a plusieurs correspondances, seule la dernière correspondance est renvoyée.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)          # The entire match
'Isaac Newton'
>>> m.group(1)          # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)          # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)       # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

Si l'expression rationnelle utilise la syntaxe `(?P<name>...)`, les arguments `groupN` peuvent alors aussi être des chaînes identifiant les groupes par leurs noms. Si une chaîne donnée en argument n'est pas utilisée comme nom de groupe dans le motif, une exception `IndexError` est levée.

Un exemple modérément compliqué :

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Les groupes nommés peuvent aussi être référencés par leur index :

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

Si un groupe a plusieurs correspondances, seule la dernière est accessible :

```
>>> m = re.match(r"(.+)", "a1b2c3") # Matches 3 times.
>>> m.group(1)                       # Returns only the last match.
'c3'
```

`match.__getitem__(g)`

Cela est identique à `m.group(g)`. Cela permet un accès plus facile à un groupe individuel depuis une correspondance :

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]          # The entire match
'Isaac Newton'
>>> m[1]          # The first parenthesized subgroup.
'Isaac'
>>> m[2]          # The second parenthesized subgroup.
'Newton'
```

Nouveau dans la version 3.6.

`match.groups (default=None)`

Renvoie un *tuple* contenant tous les sous-groupes de la correspondance, de 1 jusqu'au nombre de groupes dans le motif. L'argument `default` est utilisé pour les groupes sans correspondance ; il vaut `None` par défaut.

Par exemple :

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

Si on rend la partie décimale et tout ce qui la suit optionnels, tous les groupes ne figureront pas dans la correspondance. Ces groupes sans correspondance vaudront `None` sauf si une autre valeur est donnée à l'argument *default* :

```
>>> m = re.match(r"(\d+)\.?(\\d+)?", "24")
>>> m.groups()           # Second group defaults to None.
('24', None)
>>> m.groups('0')       # Now, the second group defaults to '0'.
('24', '0')
```

`match.groupdict (default=None)`

Renvoie un dictionnaire contenant tous les sous-groupes *nommés* de la correspondance, accessibles par leurs noms. L'argument *default* est utilisé pour les groupes qui ne figurent pas dans la correspondance ; il vaut `None` par défaut. Par exemple :

```
>>> m = re.match(r"(?P<first_name>\\w+) (?P<last_name>\\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

`match.start ([group])`

`match.end ([group])`

Renvoie les indices de début et de fin de la sous-chaîne correspondant au groupe *group* ; *group* vaut par défaut zéro (pour récupérer les indices de la correspondance complète). Renvoie `-1` si *group* existe mais ne figure pas dans la correspondance. Pour un objet de correspondance *m*, et un groupe *g* qui y figure, la sous-chaîne correspondant au groupe *g* (équivalente à `m.group(g)`) est

```
m.string[m.start(g):m.end(g)]
```

Notez que `m.start(group)` sera égal à `m.end(group)` si *group* correspond à une chaîne vide. Par exemple, après `m = re.search('b(c?)', 'cba')`, `m.start(0)` vaut 1, `m.end(0)` vaut 2, `m.start(1)` et `m.end(1)` valent tous deux 2, et `m.start(2)` lève une exception *IndexError*.

Un exemple qui supprimera *remove_this* d'une adresse mail :

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

`match.span ([group])`

Pour un objet de correspondance *m*, renvoie le *tuple* (`m.start(group)`, `m.end(group)`). Notez que si *group* ne figure pas dans la correspondance, `(-1, -1)` est renvoyé. *group* vaut par défaut zéro, pour la correspondance entière.

`match.pos`

La valeur de *pos* qui a été passée à la méthode `search()` ou `match()` d'un *objet expression rationnelle*. C'est l'index dans la chaîne à partir duquel le moteur d'expressions rationnelles recherche une correspondance.

`match.endpos`

La valeur de *endpos* qui a été passée à la méthode `search()` ou `match()` d'un *objet expression rationnelle*. C'est l'index dans la chaîne que le moteur d'expressions rationnelles ne dépassera pas.

`match.lastindex`

L'index entier du dernier groupe de capture validé, ou `None` si aucun groupe ne correspondait. Par exemple, les expressions `(a)b`, `((a)(b))` et `((ab))` auront un `lastindex == 1` si appliquées à la chaîne `'ab'`, alors que l'expression `(a)(b)` aura un `lastindex == 2` si appliquée à la même chaîne.

`match.lastgroup`

Le nom du dernier groupe capturant validé, ou `None` si le groupe n'a pas de nom, ou si aucun groupe ne correspondait.

`match.re`

L'expression rationnelle dont la méthode `match()` ou `search()` a produit cet objet de correspondance.

`match.string`

La chaîne passée à `match()` ou `search()`.

6.2.5 Exemples d'expressions rationnelles

Rechercher une paire

Dans cet exemple, nous utiliserons cette fonction de facilité pour afficher les objets de correspondance sous une meilleure forme :

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Supposez que vous écriviez un jeu de poker où la main d'un joueur est représentée par une chaîne de 5 caractères avec chaque caractère représentant une carte, « a » pour l'as, « k » pour le roi (*king*), « q » pour la reine (*queen*), « j » pour le valet (*jack*), « t » pour 10 (*ten*), et les caractères de « 2 » à « 9 » représentant les cartes avec ces valeurs.

Pour vérifier qu'une chaîne donnée est une main valide, on pourrait faire comme suit :

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q")) # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e")) # Invalid.
>>> displaymatch(valid.match("akt"))   # Invalid.
>>> displaymatch(valid.match("727ak")) # Valid.
"<Match: '727ak', groups=()>"
```

La dernière main, "727ak", contenait une paire, deux cartes de la même valeur. Pour valider cela avec une expression rationnelle, on pourrait utiliser des références arrière comme :

```
>>> pair = re.compile(r".*(.)*\1")
>>> displaymatch(pair.match("717ak")) # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak")) # No pairs.
>>> displaymatch(pair.match("354aa")) # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

Pour trouver de quelle carte est composée la paire, on pourrait utiliser la méthode `group()` de l'objet de correspondance de la manière suivante :

```
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.)*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

Simuler `scanf()`

Python n'a actuellement pas d'équivalent à la fonction C `scanf()`. Les expressions rationnelles sont généralement plus puissantes, mais aussi plus verbeuses, que les chaînes de format `scanf()`. Le tableau suivant présente des expressions rationnelles plus ou moins équivalentes aux éléments de formats de `scanf()`.

Élément de <code>scanf()</code>	Expression rationnelle
<code>%c</code>	<code>.</code>
<code>%5c</code>	<code>.{5}</code>
<code>%d</code>	<code>[-+]? \d+</code>
<code>%e, %E, %f, %g</code>	<code>[-+]? (\d+ (\.\d*)? \.\d+) ([eE] [-+]? \d+)?</code>
<code>%i</code>	<code>[-+]? (0[xX] [\dA-Fa-f]+ 0[0-7]* \d+)</code>
<code>%o</code>	<code>[-+]? [0-7]+</code>
<code>%s</code>	<code>\S+</code>
<code>%u</code>	<code>\d+</code>
<code>%x, %X</code>	<code>[-+]? (0[xX])? [\dA-Fa-f]+</code>

Pour extraire le nom de fichier et les nombres depuis une chaîne comme

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

vous utiliseriez un format `scanf()` comme

```
%s - %d errors, %d warnings
```

L'expression rationnelle équivalente serait

```
(\S+) - (\d+) errors, (\d+) warnings
```

`search()` vs. `match()`

Python offre deux opérations primitives basées sur les expressions rationnelles : `re.match()` cherche une correspondance uniquement au début de la chaîne, tandis que `re.search()` en recherche une n'importe où dans la chaîne (ce que fait Perl par défaut).

Par exemple :

```
>>> re.match("c", "abcdef")      # No match
>>> re.search("c", "abcdef")     # Match
<_sre.SRE_Match object; span=(2, 3), match='c'>
```

Les expressions rationnelles commençant par `'^'` peuvent être utilisées avec `search()` pour restreindre la recherche au début de la chaîne :

```
>>> re.match("c", "abcdef")      # No match
>>> re.search("^c", "abcdef")    # No match
>>> re.search("^a", "abcdef")    # Match
<_sre.SRE_Match object; span=(0, 1), match='a'>
```

Notez cependant qu'en mode *MULTILINE*, `match()` ne recherche qu'au début de la chaîne, alors que `search()` avec une expression rationnelle commençant par `'^'` recherchera au début de chaque ligne.

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE) # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE) # Match
<_sre.SRE_Match object; span=(4, 5), match='X'>
```

Construire un répertoire téléphonique

`split()` découpe une chaîne en une liste délimitée par le motif donné. La méthode est inestimable pour convertir des données textuelles vers des structures de données qui peuvent être lues et modifiées par Python comme démontré dans l'exemple suivant qui crée un répertoire téléphonique.

Premièrement, voici l'entrée. Elle provient normalement d'un fichier, nous utilisons ici une chaîne à guillemets triples :

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

Les entrées sont séparées par un saut de ligne ou plus. Nous convertissons maintenant la chaîne en une liste où chaque ligne non vide aura sa propre entrée :

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
 'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
 'Frank Burger: 925.541.7625 662 South Dogwood Way',
 'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finalement, on sépare chaque entrée en une liste avec prénom, nom, numéro de téléphone et adresse. Nous utilisons le paramètre `maxsplit` de `split()` parce que l'adresse contient des espaces, qui sont notre motif de séparation :

```
>>> [re.split("?: ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
 ['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
 ['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
 ['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

Le motif `?:` trouve les deux points derrière le nom de famille, pour qu'ils n'apparaissent pas dans la liste résultante. Avec un `maxsplit` de 4, nous pourrions séparer le numéro du nom de la rue :

```
>>> [re.split("?: ", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
 ['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
 ['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
 ['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

Mélanger les lettres des mots

`sub()` remplace toutes les occurrences d'un motif par une chaîne ou le résultat d'une fonction. Cet exemple le montre, en utilisant `sub()` avec une fonction qui mélange aléatoirement les caractères de chaque mot dans une phrase (à l'exception des premiers et derniers caractères) :

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reorpt yuor asnebces potlmpy.'
```

Trouver tous les adverbes

`findall()` matches *all* occurrences of a pattern, not just the first one as `search()` does. For example, if a writer wanted to find all of the adverbs in some text, they might use `findall()` in the following manner :

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

Trouver tous les adverbes et leurs positions

If one wants more information about all matches of a pattern than the matched text, `finditer()` is useful as it provides *match objects* instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs *and their positions* in some text, they would use `finditer()` in the following manner :

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

Notation brutes de chaînes

La notation brute de chaînes (`r"text"`) garde saines les expressions rationnelles. Sans elle, chaque *backslash* (`'\'`) dans une expression rationnelle devrait être préfixé d'un autre *backslash* pour l'échapper. Par exemple, les deux lignes de code suivantes sont fonctionnellement identiques :

```
>>> re.match(r"\W(.)\1\W", " ff ")
<_sre.SRE_Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<_sre.SRE_Match object; span=(0, 4), match=' ff '>
```

Pour rechercher un *backslash* littéral, il faut l'échapper dans l'expression rationnelle. Avec la notation brute, cela signifie `r"\"`. Sans elle, il faudrait utiliser `"\\\"`, faisant que les deux lignes de code suivantes sont fonctionnellement identiques :

```
>>> re.match(r"\\", r"\\")
<_sre.SRE_Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<_sre.SRE_Match object; span=(0, 1), match='\\'>
```

Écrire un analyseur lexical

Un **analyseur lexical** ou **scanner** analyse une chaîne pour catégoriser les groupes de caractères. C'est une première étape utile dans l'écriture d'un compilateur ou d'un interpréteur.

Les catégories de texte sont spécifiées par des expressions rationnelles. La technique est de les combiner dans une unique expression rationnelle maîtresse, et de boucler sur les correspondances successives :

```
import collections
import re

Token = collections.namedtuple('Token', ['type', 'value', 'line', 'column'])

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'), # Integer or decimal number
        ('ASSIGN',   r':='),          # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),   # Identifiers
        ('OP',       r'[+ \-*/]'),    # Arithmetic operators
        ('NEWLINE',  r'\n'),          # Line endings
        ('SKIP',     r'[ \t]+'),      # Skip over spaces and tabs
        ('MISMATCH', r'.'),           # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
```

(suite sur la page suivante)

(suite de la page précédente)

```
'''  
  
for token in tokenize(statements):  
    print(token)
```

L'analyseur produit la sortie suivante :

```
Token(type='IF', value='IF', line=2, column=4)  
Token(type='ID', value='quantity', line=2, column=7)  
Token(type='THEN', value='THEN', line=2, column=16)  
Token(type='ID', value='total', line=3, column=8)  
Token(type='ASSIGN', value=':=', line=3, column=14)  
Token(type='ID', value='total', line=3, column=17)  
Token(type='OP', value='+', line=3, column=23)  
Token(type='ID', value='price', line=3, column=25)  
Token(type='OP', value='*', line=3, column=31)  
Token(type='ID', value='quantity', line=3, column=33)  
Token(type='END', value=';', line=3, column=41)  
Token(type='ID', value='tax', line=4, column=8)  
Token(type='ASSIGN', value=':=', line=4, column=12)  
Token(type='ID', value='price', line=4, column=15)  
Token(type='OP', value='*', line=4, column=21)  
Token(type='NUMBER', value=0.05, line=4, column=23)  
Token(type='END', value=';', line=4, column=27)  
Token(type='ENDIF', value='ENDIF', line=5, column=4)  
Token(type='END', value=';', line=5, column=9)
```

6.3 difflib — Helpers for computing deltas

Code source : [Lib/difflib.py](#)

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce difference information in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the *filecmp* module.

class difflib.SequenceMatcher

This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are *hashable*. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershelp under the hyperbolic name « gestalt pattern matching. » The idea is to find the longest contiguous matching subsequence that contains no « junk » elements; these « junk » elements are ones that are uninteresting in some sense, such as blank lines or whitespace. (Handling junk is an extension to the Ratcliff and Obershelp algorithm.) The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that « look right » to people.

Timing : The basic Ratcliff-Obershelp algorithm is cubic time in the worst case and quadratic time in the expected case. *SequenceMatcher* is quadratic time for the worst case and has expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

Automatic junk heuristic : *SequenceMatcher* supports a heuristic that automatically treats certain sequence items as junk. The heuristic counts how many times each individual item appears in the sequence. If an item's duplicates (after the first one) account for more than 1% of the sequence and the sequence is at least 200 items long, this item is marked as « popular » and is treated as junk for the purpose of sequence matching. This heuristic can be turned off by setting the *autojunk* argument to *False* when creating the *SequenceMatcher*.

Nouveau dans la version 3.2 : The *autojunk* parameter.

class `difflib.Differ`

This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses *SequenceMatcher* both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a *Differ* delta begins with a two-letter code :

Code	Signification
'- '	line unique to sequence 1
'+ '	line unique to sequence 2
' ' '	line common to both sequences
'? '	line not present in either input sequence

Lines beginning with “?” attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain tab characters.

class `difflib.HtmlDiff`

This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is :

__init__ (*tabsize=8, wrapcolumn=None, linejunk=None, charjunk=IS_CHARACTER_JUNK*)

Initializes instance of *HtmlDiff*.

tabsize is an optional keyword argument to specify tab stop spacing and defaults to 8.

wrapcolumn is an optional keyword to specify column number where lines are broken and wrapped, defaults to None where lines are not wrapped.

linejunk and *charjunk* are optional keyword arguments passed into *ndiff()* (used by *HtmlDiff* to generate the side by side HTML differences). See *ndiff()* documentation for argument default values and descriptions.

The following methods are public :

make_file (*fromlines, tolines, fromdesc="", todesc="", context=False, numlines=5, *, charset='utf-8'*)

Compares *fromlines* and *toline*s (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.

fromdesc and *todesc* are optional keyword arguments to specify from/to file column header strings (both default to an empty string).

context and *numlines* are both optional keyword arguments. Set *context* to *True* when contextual differences are to be shown, else the default is *False* to show the full files. *numlines* defaults to 5. When *context* is *True* *numlines* controls the number of context lines which surround the difference highlights. When *context* is *False* *numlines* controls the number of lines which are shown before a difference highlight when using the « next » hyperlinks (setting to zero would cause the « next » hyperlinks to place the next difference highlight at the top of the browser without any leading context).

Modifié dans la version 3.5 : *charset* keyword-only argument was added. The default charset of HTML document changed from 'ISO-8859-1' to 'utf-8'.

make_table (*fromlines, tolines, fromdesc="", todesc="", context=False, numlines=5*)

Compares *fromlines* and *toline*s (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.

The arguments for this method are the same as those for the *make_file()* method.

Tools/scripts/diff.py is a command-line front-end to this class and contains a good example of its use.

difflib.context_diff (*a, b, fromfile="", tofile="", fromfiledate="", tofiledate="", n=3, lineterm='\n'*)

Compare *a* and *b* (lists of strings); return a delta (a *generator* generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `***` or `---`) are created with a trailing newline. This is helpful so that inputs created from `io.IOBase.readlines()` result in diffs that are suitable for use with `io.IOBase.writelines()` since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the `lineterm` argument to `" "` so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for `fromfile`, `tofile`, `fromfiledate`, and `tofiledate`. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py', tofile=
↪ 'after.py'))
*** before.py
--- after.py
*****
*** 1,4 ****
! bacon
! eggs
! ham
! guido
--- 1,4 ----
! python
! eggy
! hamster
! guido
```

See [A command-line interface to `diff`](#) for a more detailed example.

`difflib.get_close_matches(word, possibilities, n=3, cutoff=0.6)`

Return a list of the best « good enough » matches. `word` is a sequence for which close matches are desired (typically a string), and `possibilities` is a list of sequences against which to match `word` (typically a list of strings).

Optional argument `n` (default 3) is the maximum number of close matches to return; `n` must be greater than 0.

Optional argument `cutoff` (default 0.6) is a float in the range [0, 1]. Possibilities that don't score at least that similar to `word` are ignored.

The best (no more than `n`) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

`difflib.ndiff(a, b, linejunk=None, charjunk=IS_CHARACTER_JUNK)`

Compare `a` and `b` (lists of strings); return a *Differ*-style delta (a *generator* generating the delta lines).

Optional keyword parameters `linejunk` and `charjunk` are filtering functions (or `None`) :

linejunk : A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is `None`. There is also a module-level function `IS_LINE_JUNK()`, which filters out lines without visible characters, except for at most one pound character (`'#'`) – however the underlying *SequenceMatcher* class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than using this function.

charjunk : A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function `IS_CHARACTER_JUNK()`, which filters out whitespace characters (a blank or tab; it's a bad idea to include newline in this!).

`Tools/scripts/ndiff.py` is a command-line front-end to this function.

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

`difflib.restore(sequence, which)`

Return one of the two sequences that generated a delta.

Given a *sequence* produced by `Differ.compare()` or `ndiff()`, extract lines originating from file 1 or 2 (parameter *which*), stripping off line prefixes.

Example :

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join	restore(diff, 1)), end="")
one
two
three
>>> print(''.join	restore(diff, 2)), end="")
ore
tree
emu
```

`difflib.unified_diff(a, b, fromfile="", tofile="", fromfiledate="", tofiledate="", n=3, lineterm='\n')`

Compare *a* and *b* (lists of strings); return a delta (a *generator* generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in an inline style (instead of separate before/after blocks). The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with ---, +++, or @@) are created with a trailing newline. This is helpful so that inputs created from `io.IOBase.readlines()` result in diffs that are suitable for use with `io.IOBase.writelines()` since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to "" so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile=
↪ 'after.py'))
--- before.py
```

(suite sur la page suivante)

(suite de la page précédente)

```
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
  guido
```

See *A command-line interface to difflib* for a more detailed example.

`difflib.diff_bytes(dfunc, a, b, fromfile=b'', tofile=b'', fromfiledate=b'', tofiledate=b'', n=3, line-term=b'\n')`

Compare *a* and *b* (lists of bytes objects) using *dfunc*; yield a sequence of delta lines (also bytes) in the format returned by *dfunc*. *dfunc* must be a callable, typically either `unified_diff()` or `context_diff()`.

Allows you to compare data with unknown or inconsistent encoding. All inputs except *n* must be bytes objects, not str. Works by losslessly converting all inputs (except *n*) to str, and calling `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`. The output of *dfunc* is then converted back to bytes, so the delta lines that you receive have the same unknown/inconsistent encodings as *a* and *b*.

Nouveau dans la version 3.5.

`difflib.IS_LINE_JUNK(line)`

Return true for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single '#', otherwise it is not ignorable. Used as a default for parameter *linejunk* in `ndiff()` in older versions.

`difflib.IS_CHARACTER_JUNK(ch)`

Return true for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in `ndiff()`.

Voir aussi :

Pattern Matching : The Gestalt Approach Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzner. This was published in *Dr. Dobbs' Journal* in July, 1988.

6.3.1 SequenceMatcher Objects

The `SequenceMatcher` class has this constructor :

class `difflib.SequenceMatcher(isjunk=None, a='', b='', autojunk=True)`

Optional argument *isjunk* must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is « junk » and should be ignored. Passing `None` for *isjunk* is equivalent to passing `lambda x: 0`; in other words, no elements are ignored. For example, pass :

```
lambda x: x in " \t"
```

if you're comparing lines as sequences of characters, and don't want to synch up on blanks or hard tabs.

The optional arguments *a* and *b* are sequences to be compared; both default to empty strings. The elements of both sequences must be *hashable*.

The optional argument *autojunk* can be used to disable the automatic junk heuristic.

Nouveau dans la version 3.2 : The *autojunk* parameter.

`SequenceMatcher` objects get three data attributes : *bjunk* is the set of elements of *b* for which *isjunk* is `True`; *bpopular* is the set of non-junk elements considered popular by the heuristic (if it is not disabled); *b2j* is a dict mapping the remaining elements of *b* to a list of positions where they occur. All three are reset whenever *b* is reset with `set_seqs()` or `set_seq2()`.

Nouveau dans la version 3.2 : The *bjunk* and *bpopular* attributes.

SequenceMatcher objects have the following methods :

set_seqs (*a*, *b*)

Set the two sequences to be compared.

SequenceMatcher computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use *set_seq2()* to set the commonly used sequence once and call *set_seq1()* repeatedly, once for each of the other sequences.

set_seq1 (*a*)

Set the first sequence to be compared. The second sequence to be compared is not changed.

set_seq2 (*b*)

Set the second sequence to be compared. The first sequence to be compared is not changed.

find_longest_match (*alo*, *ahi*, *blo*, *bhi*)

Find longest matching block in *a*[*alo*:*ahi*] and *b*[*blo*:*bhi*].

If *isjunk* was omitted or None, *find_longest_match()* returns (*i*, *j*, *k*) such that *a*[*i*:*i*+*k*] is equal to *b*[*j*:*j*+*k*], where *alo* ≤ *i* ≤ *i*+*k* ≤ *ahi* and *blo* ≤ *j* ≤ *j*+*k* ≤ *bhi*. For all (*i'*, *j'*, *k'*) meeting those conditions, the additional conditions *k* ≥ *k'*, *i* ≤ *i'*, and if *i* == *i'*, *j* ≤ *j'* are also met. In other words, of all maximal matching blocks, return one that starts earliest in *a*, and of all those maximal matching blocks that start earliest in *a*, return the one that starts earliest in *b*.

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

If *isjunk* was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here's the same example as before, but considering blanks to be junk. That prevents ' abcd' from matching the ' abcd' at the tail end of the second sequence directly. Instead only the 'abcd' can match, and matches the leftmost 'abcd' in the second sequence :

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns (*alo*, *blo*, 0).

This method returns a *named tuple* *Match*(*a*, *b*, *size*).

get_matching_blocks ()

Return list of triples describing non-overlapping matching subsequences. Each triple is of the form (*i*, *j*, *n*), and means that *a*[*i*:*i*+*n*] == *b*[*j*:*j*+*n*]. The triples are monotonically increasing in *i* and *j*.

The last triple is a dummy, and has the value (*len*(*a*), *len*(*b*), 0). It is the only triple with *n* == 0. If (*i*, *j*, *n*) and (*i'*, *j'*, *n'*) are adjacent triples in the list, and the second is not the last triple in the list, then *i*+*n* < *i'* or *j*+*n* < *j'*; in other words, adjacent triples always describe non-adjacent equal blocks.

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes ()

Return list of 5-tuples describing how to turn *a* into *b*. Each tuple is of the form (*tag*, *i1*, *i2*, *j1*, *j2*). The first tuple has *i1* == *j1* == 0, and remaining tuples have *i1* equal to the *i2* from the preceding tuple, and, likewise, *j1* equal to the previous *j2*.

The *tag* values are strings, with these meanings :

Valeur	Signification
'replace'	<code>a[i1:i2]</code> should be replaced by <code>b[j1:j2]</code> .
'delete'	<code>a[i1:i2]</code> should be deleted. Note that <code>j1 == j2</code> in this case.
'insert'	<code>b[j1:j2]</code> should be inserted at <code>a[i1:i1]</code> . Note that <code>i1 == i2</code> in this case.
'equal'	<code>a[i1:i2] == b[j1:j2]</code> (the sub-sequences are equal).

Par exemple :

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]      'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]      'cd' --> 'cd'
insert    a[6:6] --> b[5:6]      '' --> 'f'
```

get_grouped_opcodes (*n*=3)

Return a *generator* of groups with up to *n* lines of context.

Starting with the groups returned by `get_opcodes()`, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

The groups are returned in the same format as `get_opcodes()`.

ratio ()

Return a measure of the sequences' similarity as a float in the range [0, 1].

Where T is the total number of elements in both sequences, and M is the number of matches, this is $2.0 * M / T$. Note that this is 1.0 if the sequences are identical, and 0.0 if they have nothing in common.

This is expensive to compute if `get_matching_blocks()` or `get_opcodes()` hasn't already been called, in which case you may want to try `quick_ratio()` or `real_quick_ratio()` first to get an upper bound.

quick_ratio ()

Return an upper bound on `ratio()` relatively quickly.

real_quick_ratio ()

Return an upper bound on `ratio()` very quickly.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although `quick_ratio()` and `real_quick_ratio()` are always at least as large as `ratio()` :

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.3.2 SequenceMatcher Examples

This example compares two strings, considering blanks to be « junk » :

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                       "private Thread currentThread;",
...                       "private volatile Thread currentThread;")
```

`ratio()` returns a float in $[0, 1]$, measuring the similarity of the sequences. As a rule of thumb, a `ratio()` value over 0.6 means the sequences are close matches :

```
>>> print(round(s.ratio(), 3))
0.866
```

If you're only interested in where the sequences match, `get_matching_blocks()` is handy :

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by `get_matching_blocks()` is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is 0.

If you want to know how to change the first sequence into the second, use `get_opcodes()` :

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
equal a[8:29] b[17:38]
```

Voir aussi :

- The `get_close_matches()` function in this module which shows how simple code building on `SequenceMatcher` can be used to do useful work.
- [Simple version control recipe](#) for a small application built with `SequenceMatcher`.

6.3.3 Differ Objects

Note that `Differ`-generated deltas make no claim to be **minimal** diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The `Differ` class has this constructor :

```
class difflib.Differ (linejunk=None, charjunk=None)
```

Optional keyword parameters `linejunk` and `charjunk` are for filter functions (or `None`) :

`linejunk` : A function that accepts a single string argument, and returns true if the string is junk. The default is `None`, meaning that no line is considered junk.

`charjunk` : A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is `None`, meaning that no character is considered junk.

These junk-filtering functions speed up matching to find differences and do not cause any differing lines or characters to be ignored. Read the description of the `find_longest_match()` method's `isjunk` parameter for an explanation.

`Differ` objects are used (deltas generated) via a single method :

compare (*a*, *b*)

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the `readlines()` method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the `writelines()` method of a file-like object.

6.3.4 Differ Example

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the `readlines()` method of file-like objects) :

```
>>> text1 = ''' 1. Beautiful is better than ugly.
... 2. Explicit is better than implicit.
... 3. Simple is better than complex.
... 4. Complex is better than complicated.
... '''.splitlines(keepends=True)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = ''' 1. Beautiful is better than ugly.
... 3. Simple is better than complex.
... 4. Complicated is better than complex.
... 5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

Next we instantiate a `Differ` object :

```
>>> d = Differ()
```

Note that when instantiating a `Differ` object we may pass functions to filter out line and character « junk. » See the `Differ()` constructor for details.

Finally, we compare the two :

```
>>> result = list(d.compare(text1, text2))
```

`result` is a list of strings, so let's pretty-print it :

```
>>> from pprint import pprint
>>> pprint(result)
[' 1. Beautiful is better than ugly.\n',
'- 2. Explicit is better than implicit.\n',
'- 3. Simple is better than complex.\n',
'+ 3. Simple is better than complex.\n',
'? ++\n',
'- 4. Complex is better than complicated.\n',
'? ^ ---- ^\n',
'+ 4. Complicated is better than complex.\n',
'? ++++ ^ ^\n',
'+ 5. Flat is better than nested.\n']
```

As a single multi-line string it looks like this :

```
>>> import sys
>>> sys.stdout.writelines(result)
```

(suite sur la page suivante)

(suite de la page précédente)

```

1. Beautiful is better than ugly.
- 2. Explicit is better than implicit.
- 3. Simple is better than complex.
+ 3.   Simple is better than complex.
?    ++
- 4. Complex is better than complicated.
?      ^          ---- ^
+ 4. Complicated is better than complex.
?      ++++ ^          ^
+ 5. Flat is better than nested.

```

6.3.5 A command-line interface to difflib

This example shows how to use `difflib` to create a `diff`-like utility. It is also contained in the Python source distribution, as `Tools/scripts/diff.py`.

```

#!/usr/bin/env python3
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                              timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                              '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('--fromfile')
    parser.add_argument('--tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

```

(suite sur la page suivante)

(suite de la page précédente)

```

fromdate = file_mtime(fromfile)
todate = file_mtime(tofile)
with open(fromfile) as ff:
    fromlines = ff.readlines()
with open(tofile) as tf:
    tolines = tf.readlines()

if options.u:
    diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate,
→todate, n=n)
elif options.n:
    diff = difflib.ndiff(fromlines, tolines)
elif options.m:
    diff = difflib.HtmlDiff().make_file(fromlines, tolines, fromfile, tofile,
→context=options.c, numlines=n)
else:
    diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate,
→todate, n=n)

sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()

```

6.4 textwrap — Encapsulation et remplissage de texte

Source code : [Lib/textwrap.py](#)

Le module `textwrap` fournit quelques fonctions pratiques, comme `TextWrapper`, la classe qui fait tout le travail. Si vous ne faites que formater ou ajuster une ou deux chaînes de texte, les fonctions de commodité devraient être assez bonnes; sinon, nous vous conseillons d'utiliser une instance de `TextWrapper` pour une meilleure efficacité.

`textwrap.wrap(text, width=70, **kwargs)`

Reformate le paragraphe simple dans `text` (une chaîne de caractères) de sorte que chaque ligne ait au maximum *largeur* caractères. Renvoie une liste de lignes de sortie, sans ligne vide ou caractère de fin de ligne à la fin.

Les arguments par mot-clé optionnels correspondent aux attributs d'instance de `TextWrapper`, documentés ci-dessous. `width` vaut 70 par défaut.

Consultez la méthode `TextWrapper.wrap()` pour plus de détails sur le comportement de `wrap()`.

`textwrap.fill(text, width=70, **kwargs)`

Formate le paragraphe unique dans `text` et renvoie une seule chaîne dont le contenu est le paragraphe formaté. `fill()` est un raccourci pour

```
"\n".join(wrap(text, ...))
```

En particulier, `fill()` accepte exactement les mêmes arguments par mot-clé que `wrap()`.

`textwrap.shorten(text, width, **kwargs)`

Réduit et tronque le `text` donné pour l'adapter à la *largeur* donnée.

Tout d'abord, les espaces dans `text` sont réduites (toutes les espaces blancs sont remplacées par des espaces simples). Si le résultat tient dans la `width`, il est renvoyé. Sinon, suffisamment de mots sont supprimés en fin de chaîne de manière à ce que les mots restants plus le placeholder tiennent dans `width`:

```
>>> textwrap.shorten("Hello world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

Les arguments par mot-clé optionnels correspondent aux attributs d'instance de `TextWrapper`, documentés ci-dessous. Notez que l'espace blanc est réduit avant que le texte ne soit passé à la fonction `fill()` de `TextWrapper`, donc changer la valeur de `tabsize`, `expand_tabs`, `drop_whitespace`, et `replace_whitespace` est sans effet.

Nouveau dans la version 3.4.

`textwrap.dedent(text)`

Supprime toutes les espaces en de tête de chaque ligne dans `text`.

Ceci peut être utilisé pour aligner les chaînes à trois guillemets avec le bord gauche de l'affichage, tout en les présentant sous forme indentée dans le code source.

Notez que les tabulations et les espaces sont traitées comme des espaces, mais qu'elles ne sont pas égales : les lignes " hello" et "\thello" sont considérées comme n'ayant pas d'espaces de tête communes.

Par exemple :

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
        world
    '''
    print(repr(s))          # prints '    hello\n        world\n    '
    print(repr(dedent(s)))  # prints 'hello\n    world\n'
```

`textwrap.indent(text, prefix, predicate=None)`

Ajoute `prefix` au début des lignes sélectionnées dans `text`.

Les lignes sont séparées en appelant `text.splitlines(True)`.

Par défaut, `prefix` est ajouté à toutes les lignes qui ne sont pas constituées uniquement d'espaces (y compris les fins de ligne).

Par exemple :

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, ' ')
'  hello\n\n \n  world'
```

L'argument optionnel `predicate` peut être utilisé pour contrôler quelles lignes sont en retrait. Par exemple, il est facile d'ajouter `prefix` aux lignes vides et aux lignes blanches seulement :

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

Nouveau dans la version 3.3.

`wrap()`, `fill()` et `shorten()` travaillent en créant une instance `TextWrapper` et en appelant une méthode unique sur celle-ci. Cette instance n'est pas réutilisée, donc pour les applications qui traitent plusieurs chaînes de texte en utilisant `wrap()` et/ou `fill()`, il peut être plus efficace de créer votre propre objet `TextWrapper`.

Le formatage du texte s'effectue en priorité sur les espaces puis juste après les traits d'union dans des mots séparés

par des traits d'union ; ce n'est qu'alors que les mots longs seront cassés si nécessaire, à moins que `TextWrapper.break_long_words` soit défini sur `False`.

class `textwrap.TextWrapper` (***kwargs*)

Le constructeur `TextWrapper` accepte un certain nombre d'arguments par mots-clés optionnels. Chaque argument par mot-clé correspond à un attribut d'instance, donc par exemple

```
wrapper = TextWrapper(initial_indent="* ")
```

est identique à

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

Vous pouvez réutiliser le même objet `TextWrapper` plusieurs fois et vous pouvez changer n'importe laquelle de ses options par assignation directe aux attributs d'instance entre les utilisations.

Les attributs d'instance de la classe `TextWrapper` (et les arguments par mot-clé au constructeur) sont les suivants :

width

(par défaut : 70) Longueur maximale des lignes reformatées. Tant qu'il n'y a pas de mots individuels dans le texte d'entrée plus longs que `width`, `TextWrapper` garantit qu'aucune ligne de sortie n'est plus longue que `width` caractères.

expand_tabs

(par défaut : `True`) Si `true`, alors tous les caractères de tabulation dans `text` sont transformés en espaces en utilisant la méthode `expandtabs()` de `text`.

tabsize

(par défaut : 8) Si `expand_tabs` est `true`, alors tous les caractères de tabulation dans `text` sont transformés en zéro ou plus d'espaces, selon la colonne courante et la taille de tabulation donnée.

Nouveau dans la version 3.3.

replace_whitespace

(par défaut : `True`) Si `true`, après l'expansion des tabulations mais avant le formatage, la méthode `wrap()` remplace chaque blanc par une espace unique. Les blancs remplacés sont les suivants : tabulation, nouvelle ligne, tabulation verticale, saut de page et retour chariot (`'\t\n\v\f\r'`).

Note : Si `expand_tabs` est `False` et `replace_whitespace` est vrai, chaque caractère de tabulation est remplacé par une espace unique, ce qui n'est pas la même chose que l'extension des tabulations.

Note : Si `replace_whitespace` est faux, de nouvelles lignes peuvent apparaître au milieu d'une ligne et provoquer une sortie étrange. Pour cette raison, le texte doit être divisé en paragraphes (en utilisant `str.splitlines()` ou similaire) qui sont formatés séparément.

drop_whitespace

(par défaut : `True`) Si `True`, l'espace au début et à la fin de chaque ligne (après le formatage mais avant l'indentation) est supprimée. L'espace au début du paragraphe n'est cependant pas supprimée si elle n'est pas suivie par une espace. Si les espaces en cours de suppression occupent une ligne entière, la ligne entière est supprimée.

initial_indent

(par défaut : `' '`) Chaîne qui est ajoutée à la première ligne de la sortie formatée. Elle compte pour le calcul de la longueur de la première ligne. La chaîne vide n'est pas indentée.

subsequent_indent

(par défaut : `' '`) Chaîne qui préfixe toutes les lignes de la sortie formatée sauf la première. Elle compte pour le calcul de la longueur de chaque ligne à l'exception de la première.

fix_sentence_endings

(par défaut : `Faux`) Si `true`, `TextWrapper` tente de détecter les fins de phrases et de s'assurer que les phrases

sont toujours séparées par exactement deux espaces. Ceci est généralement souhaité pour un texte en police à chasse fixe. Cependant, l'algorithme de détection de phrase est imparfait : il suppose qu'une fin de phrase consiste en une lettre minuscule suivie de l'une des lettres suivantes : '.', '!', ou '?', éventuellement suivie d'une des lettres '"' ou "'", suivie par une espace. Un problème avec cet algorithme est qu'il est incapable de détecter la différence entre « Dr » dans

```
[...] Dr. Frankenstein's monster [...]
```

et « Spot. » dans

```
[...] See Spot. See Spot run [...]
```

`fix_sentence_endings` est *False* par défaut.

Étant donné que l'algorithme de détection de phrases repose sur `string.lowercase` pour la définition de « lettres minuscules » et sur une convention consistant à utiliser deux espaces après un point pour séparer les phrases sur la même ligne, ceci est spécifique aux textes de langue anglaise.

break_long_words

(par défaut : *True*) Si *True*, alors les mots plus longs que `width` sont cassés afin de s'assurer qu'aucune ligne ne soit plus longue que `width`. Si c'est *False*, les mots longs ne sont pas cassés et certaines lignes peuvent être plus longues que `width` (les mots longs seront mis sur une ligne qui leur est propre, afin de minimiser le dépassement de `width`).

break_on_hyphens

(par défaut : *True*) Si c'est vrai, le formatage se fait de préférence sur les espaces et juste après sur les traits d'union des mots composés, comme il est d'usage en anglais. Si *False*, seuls les espaces sont considérées comme de bons endroits pour les sauts de ligne, mais vous devez définir `break_long_words` à *False* si vous voulez des mots vraiment insécables. Le comportement par défaut dans les versions précédentes était de toujours permettre de couper les mots avec trait d'union.

max_lines

(par défaut : *None*) Si ce n'est pas *None*, alors la sortie contient au maximum `max_lines` lignes, avec `placeholder` à la fin de la sortie.

Nouveau dans la version 3.4.

placeholder

(par défaut : ' ' [. . .] ') Chaîne qui apparaît à la fin du texte de sortie s'il a été tronqué.

Nouveau dans la version 3.4.

`TextWrapper` fournit également quelques méthodes publiques, analogues aux fonctions de commodité au niveau du module :

wrap (*text*)

Formate le paragraphe unique dans *text* (une chaîne de caractères) de sorte que chaque ligne ait au maximum `width` caractères. Toutes les options de formatage sont tirées des attributs d'instance de l'instance de classe `TextWrapper`. Renvoie une liste de lignes de sortie, sans nouvelles lignes finales. Si la sortie formatée n'a pas de contenu, la liste vide est renvoyée.

fill (*text*)

Formate le paragraphe unique de *text* et renvoie une seule chaîne contenant le paragraphe formaté.

6.5 unicodedata — Base de données Unicode

This module provides access to the Unicode Character Database (UCD) which defines character properties for all Unicode characters. The data contained in this database is compiled from the [UCD version 9.0.0](#).

Ce module utilise les mêmes noms et symboles définis dans l'annexe 44 du standard Unicode (*Unicode Standard Annex*), « [Unicode Character Database](#) ». Il définit les fonctions suivantes :

`unicodedata.lookup(name)`

Retrouver un caractère par nom. Si un caractère avec le nom donné est trouvé, renvoyer le caractère correspondant. S'il n'est pas trouvé, `KeyError` est levée.

Modifié dans la version 3.3 : La gestion des alias ¹ et des séquences nommées ² a été ajouté.

`unicodedata.name(chr[, default])`

Renvoie le nom assigné au caractère `chr` comme une chaîne de caractères. Si aucun nom n'est défini, `default` est renvoyé, ou, si ce dernier n'est pas renseigné `ValueError` est levée.

`unicodedata.decimal(chr[, default])`

Renvoie la valeur décimale assignée au caractère `chr` comme un entier. Si aucune valeur de ce type n'est définie, `default` est renvoyé, ou, si ce dernier n'est pas renseigné, `ValueError` est levée.

`unicodedata.digit(chr[, default])`

Renvoie le chiffre assigné au caractère `chr` comme un entier. Si aucune valeur de ce type n'est définie, `default` est renvoyé, ou, si ce dernier n'est pas renseigné, `ValueError` est levée.

`unicodedata.numeric(chr[, default])`

Renvoie la valeur numérique assignée au caractère `chr` comme un entier. Si aucune valeur de ce type n'est définie, `default` est renvoyé, ou, si ce dernier n'est pas renseigné, `ValueError` est levée.

`unicodedata.category(chr)`

Renvoie la catégorie générale assignée au caractère `chr` comme une chaîne de caractères.

`unicodedata.bidirectional(chr)`

Renvoie la classe bidirectionnelle assignée au caractère `chr` comme une chaîne de caractères. Si aucune valeur de ce type n'est définie, une chaîne de caractères vide est renvoyée.

`unicodedata.combining(chr)`

Renvoie la classe de combinaison canonique assignée au caractère `chr` comme un entier. Envoie 0 si aucune classe de combinaison n'est définie.

`unicodedata.east_asian_width(chr)`

Renvoie la largeur est-asiatique assignée à un caractère `chr` comme une chaîne de caractères.

`unicodedata.mirrored(chr)`

Renvoie la propriété miroir assignée au caractère `chr` comme un entier. Renvoie 1 si le caractère a été identifié comme un caractère « réfléchi » dans du texte bidirectionnel, sinon 0.

`unicodedata.decomposition(chr)`

Renvoie le tableau associatif de décomposition de caractère assigné au caractère `chr` comme une chaîne de caractères. Une chaîne de caractère vide est renvoyée dans le cas où aucun tableau associatif de ce type n'est défini.

`unicodedata.normalize(form, unistr)`

Renvoie la forme normale `form` de la chaîne de caractère Unicode `unistr`. Les valeurs valides de `form` sont `NFC`, `NFKC`, `NFD`, et `NFKD`.

1. <http://www.unicode.org/Public/9.0.0/ucd/NameAliases.txt>

2. <http://www.unicode.org/Public/9.0.0/ucd/NamedSequences.txt>

Le standard Unicode définit les différentes variantes de normalisation d'une chaîne de caractères Unicode en se basant sur les définitions d'équivalence canonique et d'équivalence de compatibilité. En Unicode, plusieurs caractères peuvent être exprimés de différentes façons. Par exemple, le caractère *U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA)* peut aussi être exprimé comme la séquence *U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA)*.

Pour chaque caractère, il existe deux formes normales : la forme normale C et la forme normale D. La forme normale D (NFD) est aussi appelée décomposition canonique, et traduit chaque caractère dans sa forme décomposée. La forme normale C (NFC) applique d'abord la décomposition canonique, puis compose à nouveau les caractères pré-combinés.

En plus de ces deux formes, il existe deux formes nominales basées sur l'équivalence de compatibilité. En Unicode, certains caractères sont gérés alors qu'ils sont normalement unifiés avec d'autres caractères. Par exemple, *U+2160 (ROMAN NUMERAL ONE)* est vraiment la même chose que *U+0049 (LATIN CAPITAL LETTER I)*. Cependant, ce caractère est supporté par souci de compatibilité avec les jeux de caractères existants (par exemple *gb2312*).

La forme normale KD (NFKD) applique la décomposition de compatibilité, c'est-à-dire remplacer les caractères de compatibilités avec leurs équivalents. La forme normale KC (NFKC) applique d'abord la décomposition de compatibilité, puis applique la composition canonique.

Même si deux chaînes de caractères Unicode sont normalisées et ont la même apparence pour un lecteur humain, si un a des caractères combinés et l'autre n'en a pas, elles peuvent ne pas être égales lors d'une comparaison.

De plus, ce module expose la constante suivante :

`unicodedata.unicdata_version`

La version de la base de données Unicode utilisée dans ce module.

`unicodedata.ucd_3_2_0`

Ceci est un objet qui a les mêmes méthodes que le module, mais qui utilise la version 3.2 de la base de données Unicode, pour les applications qui nécessitent cette version spécifique de la base de données Unicode (comme l'IDNA).

Exemples :

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A') # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

Notes

6.6 stringprep — Préparation des chaînes de caractères internet

Code source : [Lib/stringprep.py](#)

Nommer les différentes choses d'internet (comme les hôtes) amène souvent au besoin de comparer ces identifiants, ce qui nécessite un critère d'« égalité ». La manière dont cette comparaison est effectuée dépend du domaine d'application, c'est-à-dire si elle doit être sensible à la casse ou non. Il peut être aussi nécessaire de restreindre les identifiants possibles, pour permettre uniquement les identifiants composés de caractères « imprimables ».

La [RFC 3454](#) définit une procédure pour « préparer » des chaînes de caractères Unicode dans les protocoles internet. Avant de passer des chaînes de caractères sur le câble, elles sont traitées avec la procédure de préparation, après laquelle ils obtiennent une certaine forme normalisée. Les RFC définissent un lot de tables, qui peuvent être combinées en profils. Chaque profil doit définir quelles tables il utilise et quelles autres parties optionnelles de la procédure *stringprep* font partie du profil. Un exemple de profil *stringprep* est *nameprep*, qui est utilisé pour les noms de domaine internationalisés.

Le module *stringprep* expose uniquement les tables de la [RFC 3454](#). Comme ces tables seraient très grandes à représenter en tant que dictionnaires ou listes, le module utilise, en interne, la base de données des caractères Unicode. Le code source du module, lui-même, a été généré en utilisant l'utilitaire `mkstringprep.py`.

En conséquence, ces tables sont exposées en tant que fonctions et non en structures de données. Il y a deux types de tables dans la RFC : les ensembles et les mises en correspondance. Pour un ensemble, *stringprep* fournit la « fonction caractéristique », c'est-à-dire une fonction qui renvoie vrai si le paramètre fait partie de l'ensemble. Pour les mises en correspondance, il fournit la fonction de mise en correspondance : ayant obtenu la clé, il renvoie la valeur associée. Ci-dessous se trouve une liste de toutes les fonctions disponibles dans le module.

`stringprep.in_table_a1` (*code*)

Détermine si le code est en table A.1 (points de code non-assigné dans Unicode 3.2).

`stringprep.in_table_b1` (*code*)

Détermine si le code est en table B.1 (habituellement mis en correspondance avec rien).

`stringprep.map_table_b2` (*code*)

Renvoie la valeur correspondante à *code* selon la table B.2 (mise en correspondance pour la gestion de la casse utilisée avec *NFKC*).

`stringprep.map_table_b3` (*code*)

Renvoie la valeur correspondante à *code* dans la table B.3 (mise en correspondance pour la gestion de la casse utilisée sans normalisation).

`stringprep.in_table_c11` (*code*)

Détermine si le code est dans la table C.1.1 (caractères d'espacement ASCII).

`stringprep.in_table_c12` (*code*)

Détermine si le code est dans la table C.1.2 (caractères d'espacement non ASCII).

`stringprep.in_table_c11_c12` (*code*)

Détermine si le code est dans la table C.1 (caractères d'espacement, union de C.1.1 et C.1.2).

`stringprep.in_table_c21` (*code*)

Détermine si le code est dans la table C.2.1 (caractères de contrôle ASCII).

`stringprep.in_table_c22` (*code*)

Détermine si le code est en table C.2.2 (caractères de contrôle non ASCII).

`stringprep.in_table_c21_c22` (*code*)

Détermine si le code est dans la table C.2 (caractères de contrôle, union de C.2.1 et C.2.2).

`stringprep.in_table_c3 (code)`
Détermine si le code est en table C.3 (usage privé).

`stringprep.in_table_c4 (code)`
Détermine si le code est dans la table C.4 (points de code non-caractère).

`stringprep.in_table_c5 (code)`
Détermine si le code est en table C.5 (codes substitués).

`stringprep.in_table_c6 (code)`
Détermine si le code est dans la table C.6 (Inapproprié pour texte brut).

`stringprep.in_table_c7 (code)`
Détermine si le code est dans la table C.7 (inapproprié pour les représentations *canonics1*).

`stringprep.in_table_c8 (code)`
Détermine si le code est dans la table C.8 (change de propriétés d’affichage ou sont obsolètes).

`stringprep.in_table_c9 (code)`
Détermine si le code est dans la table C.9 (caractères de marquage).

`stringprep.in_table_d1 (code)`
Détermine si le code est en table D.1 (caractères avec propriété bidirectionnelle « R » ou « AL »).

`stringprep.in_table_d2 (code)`
Détermine si le code est dans la table D.2 (caractères avec propriété bidirectionnelle « L »).

6.7 readline — interface pour GNU *readline*

Le module `readline` définit des fonctions pour faciliter la complétion et la lecture/écriture des fichiers d’historique depuis l’interpréteur Python. Ce module peut être utilisé directement, ou depuis le module `rlcompleter`, qui gère la complétion des mots clefs dans l’invite de commande interactive. Les paramétrages faits en utilisant ce module affectent à la fois le comportement de l’invite de commande interactive et l’invite de commande fournie par la fonction native `input()`.

Note : The underlying Readline library API may be implemented by the `libedit` library instead of GNU `readline`. On MacOS X the `readline` module detects which library is being used at run time.

Le fichier de configuration pour `libedit` est différent de celui de *GNU readline*. Si, dans votre programme, vous chargez les chaînes de configuration vous pouvez valider le texte `libedit` dans `readline.__doc__` pour faire la différence entre *GNU readline* et `libedit`.

L’association de touches de *readline* peut être configurée via un fichier d’initialisation, typiquement nommé `.inputrc` dans votre répertoire utilisateur. Voir [Readline Init File](#) dans le manuel GNU pour *readline* pour des informations à propos du format et de la construction autorisée de ce fichier, ainsi que les possibilités de la bibliothèque *readline* en général.

6.7.1 Fichier d'initialisation

Les fonctions suivantes se rapportent au fichier d'initialisation et à la configuration utilisateur.

`readline.parse_and_bind(string)`

Exécute la ligne d'initialisation fournie dans l'argument *string*. Cela appelle la fonction `rl_parse_and_bind()` de la bibliothèque sous-jacente.

`readline.read_init_file([filename])`

Exécute un fichier d'initialisation *readline*. Le nom de fichier par défaut est le dernier nom de fichier utilisé. Cela appelle la fonction `rl_read_init_file()` de la bibliothèque sous-jacente.

6.7.2 Tampon de ligne

Les fonctions suivantes opèrent sur le tampon de ligne :

`readline.get_line_buffer()`

Renvoie le contenu courant du tampon de ligne (`rl_line_buffer` dans la bibliothèque sous-jacente).

`readline.insert_text(string)`

Insère du texte dans le tampon de ligne à la position du curseur. Cela appelle la fonction `rl_insert_text()` de la bibliothèque sous-jacente, mais ignore la valeur de retour.

`readline.redisplay()`

Change ce qui est affiché sur l'écran pour représenter le contenu courant de la ligne de tampon. Cela appelle la fonction `rl_redisplay()` dans la bibliothèque sous-jacente.

6.7.3 Fichier d'historique

les fonctions suivantes opèrent sur un fichier d'historique :

`readline.read_history_file([filename])`

Charge un fichier d'historique de *readline*, et l'ajoute à la liste d'historique. Le fichier par défaut est `~/.history`. Cela appelle la fonction `read_history()` de la bibliothèque sous-jacente.

`readline.write_history_file([filename])`

Enregistre la liste de l'historique dans un fichier d'historique de *readline*, en écrasant un éventuel fichier existant. Le nom de fichier par défaut est `~/.history`. Cela appelle la fonction `write_history()` de la bibliothèque sous-jacente.

`readline.append_history_file(nelements[, filename])`

Ajoute les derniers objets *nelements* de l'historique dans un fichier. Le nom de fichier par défaut est `~/.history`. Le fichier doit déjà exister. Cela appelle la fonction `append_history()` de la bibliothèque sous-jacente.

Nouveau dans la version 3.5.

`readline.get_history_length()`

`readline.set_history_length(length)`

Définit ou renvoie le nombre souhaité de lignes à enregistrer dans le fichier d'historique. La fonction `write_history_file()` utilise cette valeur pour tronquer le fichier d'historique, en appelant `history_truncate_file()` de la bibliothèque sous-jacente. Les valeurs négatives impliquent une taille de fichier d'historique illimitée.

6.7.4 Liste d'historique

Les fonctions suivantes opèrent sur une liste d'historique globale :

`readline.clear_history()`
Effacer l'historique courant. Cela appelle la fonction `clear_history()` de la bibliothèque sous-jacente. La fonction Python existe seulement si Python a été compilé pour une version de la bibliothèque qui le gère.

`readline.get_current_history_length()`
Renvoie le nombre d'objets actuellement dans l'historique. (C'est différent de `get_history_length()`, qui renvoie le nombre maximum de lignes qui vont être écrites dans un fichier d'historique.)

`readline.get_history_item(index)`
Renvoie le contenu courant de l'objet d'historique à *index*. L'index de l'objet commence à 1. Cela appelle `history_get()` de la bibliothèque sous-jacente.

`readline.remove_history_item(pos)`
Supprime l'objet de l'historique défini par sa position depuis l'historique. L'index de la position commence à zéro. Cela appelle la fonction `remove_history()` de la bibliothèque sous-jacente.

`readline.replace_history_item(pos, line)`
Remplace un objet de l'historique à la position définie par *line*. L'index de la position commence à zéro. Cela appelle `replace_history_entry()` de la bibliothèque sous-jacente.

`readline.add_history(line)`
Ajoute *line* au tampon d'historique, comme si c'était la dernière ligne saisie. Cela appelle la fonction `add_history()` de la librairie sous-jacente.

`readline.set_auto_history(enabled)`
Active ou désactive les appels automatiques à la fonction `add_history()` lors de la lecture d'une entrée via *readline*. L'argument *enabled* doit être une valeur booléenne qui lorsqu'elle est vraie, active l'historique automatique, et qui lorsqu'elle est fausse, désactive l'historique automatique.
Nouveau dans la version 3.6.
CPython implementation detail : Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.7.5 Fonctions de rappel au démarrage

`readline.set_startup_hook([function])`
Définit ou supprime la fonction invoquée par la fonction de retour `rl_startup_hook` de la bibliothèque sous-jacente. Si *function* est spécifié, il est utilisé en tant que nouvelle fonction de rappel ; si omis ou `None`, toute fonction déjà installée est supprimée. La fonction de rappel est appelée sans arguments juste avant que *readline* affiche la première invite de commande.

`readline.set_pre_input_hook([function])`
Définit ou supprime la fonction invoquée par la fonction de retour `rl_pre_input_hook` de la bibliothèque sous-jacente. Si *function* est spécifié, il sera utilisé par la nouvelle fonction de rappel ; si omis ou `None`, toute fonction déjà installée est supprimée. La fonction de rappel est appelée sans arguments après que la première invite de commande ait été affichée et juste avant que *readline* commence à lire les caractères saisis. Cette fonction existe seulement si Python a été compilé pour une version de la bibliothèque qui le gère.

6.7.6 Complétion

Les fonctions suivantes relatent comment implémenter une fonction de complétion d'un mot spécifique. C'est typiquement déclenché par la touche Tab, et peut suggérer et automatiquement compléter un mot en cours de saisie. Par défaut, Readline est configuré pour être utilisé par `rlcompleter` pour compléter les mots clefs de Python pour l'interpréteur interactif. Si le module `readline` doit être utilisé avec une complétion spécifique, un ensemble de mots délimiteurs doivent être définis.

`readline.set_completer([function])`

Définit ou supprime la fonction de complétion. Si *function* est spécifié, il sera utilisé en tant que nouvelle fonction de complétion; si omis ou `None`, toute fonction de complétion déjà installé est supprimé. La fonction de complétion est appelée telle que `function(text, state)`, pour *state* valant 0, 1, 2, ..., jusqu'à ce qu'elle renvoie une valeur qui n'est pas une chaîne de caractères. Elle doit renvoyer les prochaines complétions possibles commençant par *text*.

La fonction de complétion installée est invoquée par la fonction de retour `entry_func` passée à `rl_completion_matches()` de la bibliothèque sous-jacente. La chaîne de caractère *text* va du premier paramètres vers la fonction de retour `rl_attempted_completion_function` de la bibliothèque sous-jacente.

`readline.get_completer()`

Récupère la fonction de complétion, ou `None` si aucune fonction de complétion n'a été définie.

`readline.get_completion_type()`

Récupère le type de complétion essayé. Cela renvoie la variable `rl_completion_type` dans la bibliothèque sous-jacente en tant qu'entier.

`readline.get_begidx()`

`readline.get_endidx()`

Récupère l'indexe de début ou de fin d'un contexte de complétion. Ces indexes sont les arguments *start* et *end* passés à la fonction de retour `rl_attempted_completion_function` de la bibliothèque sous-jacente.

`readline.set_completer_delims(string)`

`readline.get_completer_delims()`

Définit ou récupère les mots délimitants pour la complétion. Ceux-ci déterminent le début du mot devant être considéré pour la complétion (le contexte de la complétion). Ces fonctions accèdent à la variable `rl_completer_word_break_characters` de la bibliothèque sous-jacente.

`readline.set_completion_display_matches_hook([function])`

Définit ou supprime la fonction d'affichage de la complétion. Si *function* est spécifié, il sera utilisé en tant que nouvelle fonction d'affichage de complétion; si omis ou `None`, toute fonction de complétion déjà installée est supprimée. Cela définit ou supprime la fonction de retour `rl_completion_display_matches_hook` dans la bibliothèque sous-jacente. La fonction d'affichage de complétion est appelée telle que `function(substitution, [matches], longest_match_length)` une seule fois lorsque les correspondances doivent être affichées.

6.7.7 Exemple

L'exemple suivant démontre comment utiliser les fonctions de lecture et d'écriture de l'historique du module `readline` pour charger ou sauvegarder automatiquement un fichier d'historique nommé `.python_history` depuis le répertoire d'accueil de l'utilisateur. Le code ci-dessous doit normalement être exécuté automatiquement durant une session interactive depuis le fichier de l'utilisateur `PYTHONSTARTUP`.

```
import atexit
import os
import readline
```

(suite sur la page suivante)

(suite de la page précédente)

```

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)

```

Ce code est en réalité automatiquement exécuté lorsque Python tourne en mode interactif (voir [Readline configuration](#)).

L'exemple suivant atteint le même objectif mais gère des sessions interactives concurrentes, en ajoutant seulement le nouvel historique.

```

import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)

```

L'exemple suivant étend la classe `code.InteractiveConsole` pour gérer la sauvegarde/restauration de l'historique.

```

import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):

```

(suite sur la page suivante)

(suite de la page précédente)

```

readline.set_history_length(1000)
readline.write_history_file(histfile)

```

6.8 rlcompleter — Fonction de complétion pour *GNU readline*

Code source : [Lib/rlcompleter.py](#)

Le module `rlcompleter` définit une fonction de complétion appropriée pour le module `readline` en complétant des identifiants et mots-clés Python valides.

Quand le module est importé dans une plate-forme Unix et la méthode `complete()` est configurée pour assurer la complétion de `readline`, une instance de classe `Completer` est automatiquement créée et la méthode est configurée au finisseur `complete()`

Exemple :

```

>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__        readline.insert_text(      readline.set_completer(
readline.__name__        readline.parse_and_bind(
>>> readline.

```

Le module `rlcompleter` est conçu pour être utilisé par le mode interactif de Python. À moins que Python ne soit exécuté avec l'option `-S`, le module est automatiquement importé et configuré (voir [Readline configuration](#)).

Sur les plate-formes sans `readline`, la classe `Completer` définie par ce module peut quand même être utilisée pour des fins personnalisées.

6.8.1 Objets pour la complétion (*Completer Objects*)

Les objets pour la complétion (*Completer objects* en anglais) disposent de la méthode suivante :

`Completer.complete(text, state)`

Renvoie la *state*-ième complétion pour *text*.

Si *text* ne contient pas un caractère point (`'.'`), il puise dans les noms actuellement définis dans `__main__`, `builtins` ainsi que les mots-clés (ainsi que définis par le module `keyword`)

Quand elle est appelée pour un nom qui comporte un point, elle ne tente d'évaluer que ce qui n'a pas d'effet secondaire (les fonctions ne sont pas évaluées, mais elle peut faire des appels à `__getattr__()`), jusqu'à la dernière partie, et trouve des équivalents pour le reste via la fonction `dir()`. Toute exception levée durant l'évaluation de l'expression est interceptée, mise sous silence, et `None` est renvoyé.

Services autour des Données Binaires

Les modules décrits dans ce chapitre fournissent des services élémentaires pour manipuler des données binaires. Les autres manipulations sur les données binaires, particulièrement celles en relation avec les formats de fichier et les protocoles réseaux sont décrits dans leurs propres chapitres.

Certaines bibliothèques décrites dans *Services de Manipulation de Texte* fonctionnent aussi avec soit des formats binaires compatibles ASCII (comme le module *re*) soit toutes les données binaires (comme le module *difflib*).

En complément, consultez la documentation des types natifs binaires dans *Séquences Binaires* — *bytes*, *bytearray*, *memoryview*.

7.1 *struct* — manipulation de données agrégées sous forme binaire comme une séquence d'octets

Code source : [Lib/struct.py](#)

Ce module effectue des conversions entre des valeurs Python et des structures C représentées sous la forme de *bytes* (séquences d'octets) Python. Cela permet, entre autres, de manipuler des données agrégées sous forme binaire dans des fichiers ou à travers des connecteurs réseau. Il utilise *Chaînes de spécification du format* comme description de l'agencement des structures afin de réaliser les conversions depuis et vers les valeurs Python.

Note : par défaut, le résultat de l'agrégation d'une structure C donnée comprend des octets de bourrage afin de maintenir un alignement correct des types C sous-jacents ; de la même manière, l'alignement est pris en compte lors de la dissociation. Ce comportement a été choisi de manière à ce que les octets d'une structure agrégée reproduisent exactement l'agencement en mémoire de la structure C équivalente. Pour gérer des formats de données indépendants de la plateforme ou omettre les octets implicites de bourrage, utilisez la taille et l'alignement *standard* en lieu et place de la taille et l'alignement *native* (voir *Boutisme, taille et alignement* pour les détails).

Plusieurs fonctions de *struct* (et méthodes de *Struct*) prennent un argument *buffer*. Cet argument fait référence à des objets qui implémentent le protocole tampon et qui proposent un tampon soit en lecture seule, soit en lecture-écriture.

Les types les plus courants qui utilisent cette fonctionnalité sont *bytes* et *bytearray*, mais beaucoup d'autres types qui peuvent être considérés comme des tableaux d'octets implémentent le protocole tampon ; ils peuvent ainsi être lus ou remplis depuis un objet *bytes* sans faire de copie.

7.1.1 Fonctions et exceptions

Le module définit les exceptions et fonctions suivantes :

exception `struct.error`

Exception levée à plusieurs occasions ; l'argument est une chaîne qui décrit ce qui ne va pas.

`struct.pack(fmt, v1, v2, ...)`

Return a bytes object containing the values *v1*, *v2*, ... packed according to the format string *fmt*. The arguments must match the values required by the format exactly.

`struct.pack_into(fmt, buffer, offset, v1, v2, ...)`

Pack the values *v1*, *v2*, ... according to the format string *fmt* and write the packed bytes into the writable buffer *buffer* starting at position *offset*. Note that *offset* is a required argument.

`struct.unpack(fmt, buffer)`

Unpack from the buffer *buffer* (presumably packed by `pack(fmt, ...)`) according to the format string *fmt*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes must match the size required by the format, as reflected by `calcsize()`.

`struct.unpack_from(fmt, buffer, offset=0)`

Unpack from *buffer* starting at position *offset*, according to the format string *fmt*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes, minus *offset*, must be at least the size required by the format, as reflected by `calcsize()`.

`struct.iter_unpack(fmt, buffer)`

Iteratively unpack from the buffer *buffer* according to the format string *fmt*. This function returns an iterator which will read equally-sized chunks from the buffer until all its contents have been consumed. The buffer's size in bytes must be a multiple of the size required by the format, as reflected by `calcsize()`.

Chaque itération produit un n-uplet tel que spécifié par la chaîne de format.

Nouveau dans la version 3.4.

`struct.calcsize(fmt)`

Return the size of the struct (and hence of the bytes object produced by `pack(fmt, ...)`) corresponding to the format string *fmt*.

7.1.2 Chaînes de spécification du format

Les chaînes de spécification du format servent à définir l'agencement lors de l'agrégation et la dissociation des données. Elles sont construites à partir de *Caractères de format*, qui spécifient le type de donnée à agréger-dissocier. De plus, il existe des caractères spéciaux pour contrôler *Boutisme, taille et alignement*.

Boutisme, taille et alignement

Par défaut, les types C sont représentés dans le format et le boutisme natifs de la machine ; ils sont alignés correctement en sautant des octets si nécessaire (en fonction des règles utilisées par le compilateur C).

Cependant, le premier caractère de la chaîne de format peut être utilisé pour indiquer le boutisme, la taille et l'alignement des données agrégées, conformément à la table suivante :

Caractère	Boutisme	Taille	Alignement
@	natif	natif	natif
=	natif	standard	aucun
<	petit-boutiste	standard	aucun
>	gros-boutiste	standard	aucun
!	réseau (= gros-boutiste)	standard	aucun

Si le premier caractère n'est pas dans cette liste, le module se comporte comme si '@' avait été indiqué.

Le boutisme natif est gros-boutiste ou petit-boutiste, en fonction de la machine sur laquelle s'exécute le programme. Par exemple, les Intel x86 et les AMD64 (x86-64) sont petit-boutistes ; les Motorola 68000 et les *PowerPC G5* sont gros-boutistes ; les ARM et les Intel Itanium peuvent changer de boutisme. Utilisez `sys.byteorder` pour vérifier le boutisme de votre système.

La taille et l'alignement natifs sont déterminés en utilisant l'expression `sizeof` du compilateur C. Leur valeur est toujours combinée au boutisme natif.

La taille standard dépend seulement du caractère du format ; référez-vous au tableau dans la section [Caractères de format](#).

Notez la différence entre '@' et '=' : les deux utilisent le boutisme natif mais la taille et l'alignement du dernier sont standards.

La forme '!' existe pour les têtes en l'air qui prétendent ne pas se rappeler si le boutisme réseau est gros-boutiste ou petit-boutiste.

Il n'y a pas de moyen de spécifier le boutisme contraire au boutisme natif (c'est-à-dire forcer la permutation des octets) ; utilisez le bon caractère entre '<' et '>'.

Notes :

- (1) Le bourrage (*padding* en anglais) n'est automatiquement ajouté qu'entre les membres successifs de la structure. Il n'y a pas de bourrage au début ou à la fin de la structure agrégée.
- (2) Il n'y a pas d'ajout de bourrage lorsque vous utilisez une taille et un alignement non-natifs, par exemple avec '<', '>', '=' ou '!.
- (3) Pour aligner la fin d'une structure à l'alignement requis par un type particulier, terminez le format avec le code du type voulu et une valeur de répétition à zéro. Référez-vous à [Exemples](#).

Caractères de format

Les caractères de format possèdent les significations suivantes ; la conversion entre les valeurs C et Python doit être évidente compte tenu des types concernés. La colonne « taille standard » fait référence à la taille en octets de la valeur agrégée avec l'utilisation de la taille standard (c'est-à-dire lorsque la chaîne de format commence par l'un des caractères suivants : '<', '>', '!' ou '='). Si vous utilisez la taille native, la taille de la valeur agrégée dépend de la plateforme.

Format	Type C	Type Python	Taille standard	Notes
x	octet de bourrage	pas de valeur		
c	char	<i>bytes</i> (suite d'octets) de taille 1	1	
b	signed char	<i>int</i> (entier)	1	(1),(3)
B	unsigned char	<i>int</i> (entier)	1	(3)
?	_Bool	<i>bool</i> (booléen)	1	(1)
h	short	<i>int</i> (entier)	2	(3)
H	unsigned short	<i>int</i> (entier)	2	(3)
i	int	<i>int</i> (entier)	4	(3)
I	unsigned int	<i>int</i> (entier)	4	(3)
l	long	<i>int</i> (entier)	4	(3)
L	unsigned long	<i>int</i> (entier)	4	(3)
q	long long	<i>int</i> (entier)	8	(2), (3)
Q	unsigned long long	<i>int</i> (entier)	8	(2), (3)
n	ssize_t	<i>int</i> (entier)		(4)
N	size_t	<i>int</i> (entier)		(4)
e	(7)	<i>float</i> (nombre à virgule flottante)	2	(5)
f	float	<i>float</i> (nombre à virgule flottante)	4	(5)
d	double	<i>float</i> (nombre à virgule flottante)	8	(5)
s	char[]	<i>bytes</i> (séquence d'octets)		
p	char[]	<i>bytes</i> (séquence d'octets)		
P	void *	<i>int</i> (entier)		(6)

Modifié dans la version 3.3 : ajouté la gestion des formats 'n' et 'N'.

Modifié dans la version 3.6 : ajouté la gestion du format 'e'.

Notes :

- (1) Le code de conversion '?' correspond au type `_Bool` de C99. Si ce type n'est pas disponible, il est simulé en utilisant un `char`. Dans le mode standard, il est toujours représenté par un octet.
- (2) The 'q' and 'Q' conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, `__int64`. They are always available in standard modes.
- (3) Lorsque vous essayez d'agréger un non-entier en utilisant un code de conversion pour un entier, si ce non-entier possède une méthode `__index__()` alors cette méthode est appelée pour convertir l'argument en entier avant l'agrégation.
Modifié dans la version 3.2 : utilisation de la méthode `__index__()` pour les non-entiers.
- (4) Les codes de conversion 'n' et 'N' ne sont disponibles que pour la taille native (choisie par défaut ou à l'aide du caractère de boutisme '@'). Pour la taille standard, vous pouvez utiliser n'importe quel format d'entier qui convient à votre application.
- (5) Pour les codes de conversion 'f', 'd' et 'e', la représentation agrégée utilise respectivement le format IEEE 754 *binair32*, *binair64* ou *binair16* (pour 'f', 'd' ou 'e' respectivement), quel que soit le format des nombres à virgule flottante de la plateforme.
- (6) Le caractère de format 'P' n'est disponible que pour le boutisme natif (choisi par défaut ou à l'aide du caractère '@' de boutisme). Le caractère de boutisme '=' choisit d'utiliser un petit ou un gros en fonction du système hôte. Le module *struct* ne l'interprète pas comme un boutisme natif, donc le format 'P' n'est pas disponible.
- (7) Le type IEEE 754 *binair16* « demie-précision » a été introduit en 2008 par la révision du [standard IEEE 754](#). Il comprend un bit de signe, un exposant sur 5 bits et une précision de 11 bits (dont 10 bits sont explicitement stockés) ; il peut représenter les nombres entre environ $6.1e-05$ et $6.5e+04$ avec une précision maximale. Ce type est rarement pris en charge par les compilateurs C : sur une machine courante, un *unsigned short* (entier court non signé) peut être utilisé pour le stockage mais pas pour les opérations mathématiques. Lisez la page Wikipédia (NdT : non traduite en français) [half-precision floating-point format](#) pour davantage d'informations.

Un caractère de format peut être précédé par un entier indiquant le nombre de répétitions. Par exemple, la chaîne de format `'4h'` a exactement la même signification que `'hhhh'`.

Les caractères d'espacement entre les indications de format sont ignorés ; cependant, le nombre de répétitions et le format associé ne doivent pas être séparés par des caractères d'espacement.

Pour le caractère de format `'s'`, un nombre en tête est interprété comme la longueur du *bytes* et non comme le nombre de répétitions comme pour les autres caractères de format ; par exemple, `'10s'` signifie une seule chaîne de 10 octets alors que `'10c'` signifie 10 caractères. Si aucun nombre n'est indiqué, la valeur par défaut est 1. Pour l'agrégation, la chaîne est tronquée ou bourrée avec des octets nuls pour atteindre la taille souhaitée. Pour la dissociation, l'objet *bytes* résultant possède le nombre exact d'octets spécifiés. Un cas particulier est `'0s'` qui signifie une chaîne (et une seule) vide (alors que `'0c'` signifie zéro caractère).

Lors de l'agrégation d'une valeur *x* en utilisant l'un des formats pour les entiers (`'b'`, `'B'`, `'h'`, `'H'`, `'i'`, `'I'`, `'l'`, `'L'`, `'q'`, `'Q'`), si *x* est en dehors de l'intervalle du format spécifié, une `struct.error` est levée.

Modifié dans la version 3.1 : auparavant, certains formats d'entiers absorbaient les valeurs en dehors des intervalles valides et levaient une `DeprecationWarning` au lieu d'une `struct.error`.

Le caractère de format `'p'` sert à encoder une « chaîne Pascal », c'est-à-dire une courte chaîne de longueur variable, stockée dans un *nombre défini d'octets* dont la valeur est définie par la répétition. Le premier octet stocké est la longueur de la chaîne (dans la limite maximum de 255). Les octets composant la chaîne suivent. Si la chaîne passée à `pack()` est trop longue (supérieure à la valeur de la répétition moins 1), seuls les `count-1` premiers octets de la chaîne sont stockés. Si la chaîne est plus courte que `count-1`, des octets de bourrage nuls sont insérés de manière à avoir exactement *count* octets au final. Notez que pour `unpack()`, le caractère de format `'p'` consomme *count* octets mais que la chaîne renvoyée ne peut pas excéder 255 octets.

Pour le caractère de format `'?'`, la valeur renvoyée est `True` ou `False`. Lors de l'agrégation, la valeur de vérité de l'objet argument est utilisée. La valeur agrégée est 0 ou 1 dans la représentation native ou standard et, lors de la dissociation, n'importe quelle valeur différente de zéro est renvoyée `True`.

Exemples

Note : tous les exemples présentés supposent que l'on utilise le boutisme, la taille et l'alignement natifs sur une machine gros-boutiste.

Un exemple de base d'agrégation et dissociation de trois entiers :

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

Les champs dissociés peuvent être nommés en leur assignant des variables ou en encapsulant le résultat dans un n-uplet nommé :

```
>>> record = b'raymond \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond ', serialnum=4658, school=264, gradelevel=8)
```

L'ordre des caractères de format peut avoir un impact sur la taille puisque le bourrage nécessaire pour réaliser l'alignement est différent :

```
>>> pack('ci', b'*', 0x12131415)
b'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, b'*')
b'\x12\x13\x14\x15*'
>>> calcsizes('ci')
8
>>> calcsizes('ic')
5
```

Le format suivant 'llh01' spécifie deux octets de bourrage à la fin, considérant que les entiers longs sont alignés sur des espacements de 4 octets :

```
>>> pack('llh01', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

Ceci ne fonctionne que quand la taille et l'alignement natifs sont utilisés ; la taille et l'alignement standards ne forcent aucun alignement.

Voir aussi :

Module [array](#) Stockage agrégé binaire de données homogènes.

Module [xdrlib](#) Agrégation et dissociation de données XDR.

7.1.3 Classes

Le module `struct` définit aussi le type suivant :

class `struct.Struct` (*format*)

Renvoie un nouvel objet Struct qui écrit et lit des données binaires conformément à la chaîne de format *format*. Créer une fois pour toutes un objet Struct puis appeler ses méthodes est plus efficace que d'appeler les fonctions de `struct` avec le même format puisque la chaîne de format n'est compilée qu'une seule fois.

Les objets Struct compilés gèrent les méthodes et attributs suivants :

pack (*v1*, *v2*, ...)

Identique à la fonction `pack()`, en utilisant le format compilé (`len(result)` vaut *size*).

pack_into (*buffer*, *offset*, *v1*, *v2*, ...)

Identique à la fonction `pack_into()`, en utilisant le format compilé.

unpack (*buffer*)

Identique à la fonction `unpack()`, en utilisant le format compilé. La taille du tampon *buffer* en octets doit valoir *size*.

unpack_from (*buffer*, *offset*=0)

Identical to the `unpack_from()` function, using the compiled format. The buffer's size in bytes, minus *offset*, must be at least *size*.

iter_unpack (*buffer*)

Identique à la fonction `iter_unpack()`, en utilisant le format compilé. La taille du tampon *buffer* en octets doit être un multiple de *size*.

Nouveau dans la version 3.4.

format

La chaîne de format utilisée pour construire l'objet *Struct*.

size

La taille calculée de la structure agrégée (et donc de l'objet *bytes* produit par la méthode `pack()`) correspondante à *format*.

7.2 codecs — Registre des codecs et classes de base associées

Code source : [Lib/codecs.py](#)

Ce module définit les classes de base pour les codecs (encodeurs et décodeurs) standards Python et fournit l'interface avec le registre des codecs internes à Python, qui gère le processus de recherche de codecs et de gestion des erreurs. La plupart des codecs sont des *encodeurs de texte*, qui encode du texte vers des séquences d'octets (type *bytes* de Python) mais il existe aussi des codecs qui encodent du texte vers du texte et des *bytes* vers des *bytes*. Les codecs personnalisés peuvent encoder et décoder des types arbitraires, mais l'utilisation de certaines fonctionnalités du module est restreinte aux *encodeurs de texte* ou aux codecs qui encodent vers *bytes*.

Le module définit les fonctions suivantes pour encoder et décoder à l'aide de n'importe quel codec :

`codecs.encode(obj, encoding='utf-8', errors='strict')`

Encode *obj* en utilisant le codec enregistré pour *encoding*.

Vous pouvez spécifier *errors* pour définir la façon de gérer les erreurs. Le gestionnaire d'erreurs par défaut est 'strict', ce qui veut dire qu'une erreur lors de l'encodage lève *ValueError* (ou une sous-classe spécifique du codec, telle que *UnicodeEncodeError*). Référez-vous aux *Classes de base de codecs* pour plus d'informations sur la gestion des erreurs par les codecs.

`codecs.decode(obj, encoding='utf-8', errors='strict')`

Décode *obj* en utilisant le codec enregistré pour *encoding*.

Vous pouvez spécifier *errors* pour définir la façon de gérer les erreurs. Le gestionnaire d'erreurs par défaut est 'strict', ce qui veut dire qu'une erreur lors du décodage lève *ValueError* (ou une sous-classe spécifique du codec, telle que *UnicodeDecodeError*). Référez-vous aux *Classes de base de codecs* pour plus d'informations sur la gestion des erreurs par les codecs.

Les détails complets de chaque codec peuvent être examinés directement :

`codecs.lookup(encoding)`

Recherche les informations relatives au codec dans le registre des codecs de Python et renvoie l'objet *CodecInfo* tel que défini ci-dessous.

Les encodeurs sont recherchés en priorité dans le cache du registre. S'ils n'y sont pas, la liste des fonctions de recherche enregistrées est passée en revue. Si aucun objet *CodecInfo* n'est trouvé, une *LookupError* est levée. Sinon, l'objet *CodecInfo* est mis en cache et renvoyé vers l'appelant.

class `codecs.CodecInfo` (*encode, decode, streamreader=None, streamwriter=None, incrementalencoder=None, incrementaldecoder=None, name=None*)

Les détails d'un codec trouvé dans le registre des codecs. Les arguments du constructeur sont stockés dans les attributs éponymes :

name

Le nom de l'encodeur.

encode

decode

Les fonctions d'encodage et de décodage. Ces fonctions ou méthodes doivent avoir la même interface que les méthodes *encode()* et *decode()* des instances de *Codec* (voir *Interface des codecs*). Les fonctions et méthodes sont censées fonctionner sans état interne.

incrementalencoder

incrementaldecoder

Classes d'encodeurs et de décodeurs incrémentaux ou fonctions usines. Elles doivent avoir respectivement les mêmes interfaces que celles définies par les classes de base *IncrementalEncoder* et *IncrementalDecoder*. Les codecs incrémentaux peuvent conserver des états internes.

streamwriter

streamreader

Classes d'écriture et de lecture de flux ou fonctions usines. Elles doivent avoir les mêmes interfaces que celles définies par les classes de base *StreamWriter* et *StreamReader*, respectivement. Les codecs de flux peuvent conserver un état interne.

Pour simplifier l'accès aux différents composants du codec, le module fournit les fonctions supplémentaires suivantes qui utilisent *lookup()* pour la recherche du codec :

`codecs.getencoder(encoding)`

Recherche le codec pour l'encodage *encoding* et renvoie sa fonction d'encodage.

Lève une *LookupError* si l'encodage *encoding* n'est pas trouvé.

`codecs.getdecoder(encoding)`

Recherche le codec pour l'encodage *encoding* et renvoie sa fonction de décodage.

Lève une *LookupError* si l'encodage *encoding* n'est pas trouvé.

`codecs.getincrementalencoder(encoding)`

Recherche le codec pour l'encodage *encoding* et renvoie sa classe d'encodage incrémental ou la fonction usine.

Lève une *LookupError* si l'encodage *encoding* n'est pas trouvé ou si le codec ne gère pas l'encodage incrémental.

`codecs.getincrementaldecoder(encoding)`

Recherche le codec pour l'encodage *encoding* et renvoie sa classe de décodage incrémental ou la fonction usine.

Lève une *LookupError* si l'encodage *encoding* n'est pas trouvé ou si le codec ne gère pas le décodage incrémental.

`codecs.getreader(encoding)`

Recherche le codec pour l'encodage *encoding* et renvoie sa classe *StreamReader* ou la fonction usine.

Lève une *LookupError* si l'encodage *encoding* n'est pas trouvé.

`codecs.getwriter(encoding)`

Recherche le codec pour l'encodage *encoding* et renvoie sa classe *StreamWriter* ou la fonction usine.

Lève une *LookupError* si l'encodage *encoding* n'est pas trouvé.

Les codecs personnalisés sont mis à disposition en enregistrant une fonction de recherche de codecs adaptée :

`codecs.register(search_function)`

Enregistre une fonction de recherche de codec. Il convient qu'une fonction de recherche prenne un argument, le nom de l'encodage écrit en lettres minuscules, et renvoie un objet *CodecInfo*. Si la fonction de recherche ne trouve pas un encodage donné, il convient qu'elle renvoie "None".

Note : l'enregistrement d'une fonction de recherche n'est actuellement pas réversible, ce qui peut entraîner des problèmes dans certains cas, par exemple pour les tests unitaires ou le rechargement de module.

Alors qu'il est recommandé d'utiliser la fonction native *open()* et le module associé *io* pour travailler avec des fichiers texte encodés, le présent module fournit des fonctions et classes utilitaires supplémentaires qui permettent l'utilisation d'une plus large gamme de codecs si vous travaillez avec des fichiers binaires :

`codecs.open(filename, mode='r', encoding=None, errors='strict', buffering=1)`

Ouvre un fichier encodé en utilisant le *mode* donné et renvoie une instance de *StreamReaderWriter*, permettant un encodage-décodage transparent. Le mode de fichier par défaut est 'r', ce qui signifie que le fichier est ouvert en lecture.

Note : les fichiers encodés sous-jacents sont toujours ouverts en mode binaire. Aucune conversion automatique de '\n' n'est effectuée à la lecture ou à l'écriture. L'argument *mode* peut être n'importe quel mode binaire acceptable pour la fonction native *open()* ; le 'b' est automatiquement ajouté.

encoding spécifie l'encodage à utiliser pour le fichier. Tout encodage qui encode et décode des octets (type *bytes*) est autorisé et les types de données pris en charge par les méthodes relatives aux fichiers dépendent du codec utilisé.

errors peut être spécifié pour définir la gestion des erreurs. La valeur par défaut est `'strict'`, ce qui lève une `ValueError` en cas d'erreur lors du codage.

buffering has the same meaning as for the built-in `open()` function. It defaults to line buffered.

`codecs.EncodedFile` (*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)

Renvoie une instance de `StreamRecorder`, version encapsulée de *file* qui fournit un transcodage transparent. Le fichier original est fermé quand la version encapsulée est fermée.

Les données écrites dans un fichier encapsulant sont décodées en fonction du *data_encoding* spécifié puis écrites vers le fichier original en tant que *bytes* en utilisant *file_encoding*. Les octets lus dans le fichier original sont décodés conformément à *file_encoding* et le résultat est encodé en utilisant *data_encoding*.

Si *file_encoding* n'est pas spécifié, la valeur par défaut est *data_encoding*.

errors peut être spécifié pour définir la gestion des erreurs. La valeur par défaut est `'strict'`, ce qui lève une `ValueError` en cas d'erreur lors du codage.

`codecs.iterencode` (*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

Utilise un encodeur incrémental pour encoder de manière itérative l'entrée fournie par *iterator*. Cette fonction est un *générateur*. L'argument *errors* (ainsi que tout autre argument passé par mot-clé) est transmis à l'encodeur incrémental.

Cette fonction nécessite que le codec accepte les objets texte (classe `str`) en entrée. Par conséquent, il ne prend pas en charge les encodeurs *bytes* vers *bytes* tels que `base64_codec`.

`codecs.iterdecode` (*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

Utilise un décodeur incrémental pour décoder de manière itérative l'entrée fournie par *iterator*. Cette fonction est un *générateur*. L'argument *errors* (ainsi que tout autre argument passé par mot-clé) est transmis au décodeur incrémental.

Cette fonction requiert que le codec accepte les objets *bytes* en entrée. Par conséquent, elle ne prend pas en charge les encodeurs de texte vers texte tels que `rot_13`, bien que `rot_13` puisse être utilisé de manière équivalente avec `iterencode()`.

Le module fournit également les constantes suivantes qui sont utiles pour lire et écrire les fichiers dépendants de la plateforme :

```
codecs.BOM
codecs.BOM_BE
codecs.BOM_LE
codecs.BOM_UTF8
codecs.BOM_UTF16
codecs.BOM_UTF16_BE
codecs.BOM_UTF16_LE
codecs.BOM_UTF32
codecs.BOM_UTF32_BE
codecs.BOM_UTF32_LE
```

Ces constantes définissent diverses séquences d'octets, les marques d'ordre d'octets (BOM pour *byte order mark* en anglais) Unicode pour plusieurs encodages. Elles sont utilisées dans les flux de données UTF-16 et UTF-32 pour indiquer l'ordre des octets utilisé, et dans UTF-8 comme signature Unicode. `BOM_UTF16` vaut soit `BOM_UTF16_BE`, soit `BOM_UTF16_LE` selon le boutisme natif de la plateforme, `BOM` est un alias pour `BOM_UTF16`, `BOM_LE` pour `BOM_UTF16_LE` et `BOM_BE` pour `BOM_UTF16_BE`. Les autres sont les marques BOM dans les encodages UTF-8 et UTF-32.

7.2.1 Classes de base de codecs

Le module `codecs` définit un ensemble de classes de base qui spécifient les interfaces pour travailler avec des objets codecs et qui peuvent également être utilisées comme base pour des implémentations de codecs personnalisés.

Chaque codec doit définir quatre interfaces pour être utilisable comme codec en Python : codeur sans état, décodeur sans état, lecteur de flux et écrivain de flux. Le lecteur et l'écrivain de flux réutilisent généralement l'encodeur-décodeur sans état pour implémenter les protocoles de fichiers. Les auteurs de codecs doivent également définir comment le codec gère les erreurs d'encodage et de décodage.

Gestionnaires d'erreurs

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument. The following string values are defined and implemented by all standard Python codecs :

Valeur	Signification
'strict'	Raise <code>UnicodeError</code> (or a subclass); this is the default. Implemented in <code>strict_errors()</code> .
'ignore'	Ignore the malformed data and continue without further notice. Implemented in <code>ignore_errors()</code> .

Les gestionnaires d'erreurs suivants ne s'appliquent que pour les *encodeurs de texte* :

Valeur	Signification
'replace'	Replace with a suitable replacement marker; Python will use the official U+FFFD REPLACE-CHARACTER for the built-in codecs on decoding, and “?” on encoding. Implemented in <code>replace_errors()</code> .
'xmlcharrefreplace'	Replace with the appropriate XML character reference (only for encoding). Implemented in <code>xmlcharrefreplace_errors()</code> .
'backslashreplace'	Replace with a sequence escaped by antislashes. Implémenté dans <code>backslashreplace_errors()</code> .
'namereplace'	Replace with <code>\N{...}</code> escape sequences (only for encoding). Implemented in <code>namereplace_errors()</code> .
'surrogateescape'	On decoding, replace byte with individual surrogate code ranging from U+DC80 to U+DCFF. This code will then be turned back into the same byte when the 'surrogateescape' error handler is used when encoding the data. (See PEP 383 for more.)

En plus, le gestionnaire d'erreurs suivant est spécifique aux codecs suivants :

Valeur	Codecs	Signification
'surrogatepass'	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding of surrogate codes. These codecs normally treat the presence of surrogates as an error.

Nouveau dans la version 3.1 : les gestionnaires d'erreurs 'surrogateescape' et 'surrogatepass'.

Modifié dans la version 3.4 : le gestionnaire d'erreurs 'surrogatepass' fonctionne maintenant avec les codecs utf-16* et utf-32*.

Nouveau dans la version 3.5 : le gestionnaire d'erreurs 'namereplace'.

Modifié dans la version 3.5 : le gestionnaire d'erreurs 'backslashreplace' fonctionne maintenant pour le décodage et la traduction.

L'ensemble des valeurs autorisées peut être étendu en enregistrant un nouveau gestionnaire d'erreurs nommé :

`codecs.register_error` (*name*, *error_handler*)

Register the error handling function *error_handler* under the name *name*. The *error_handler* argument will be called during encoding and decoding in case of an error, when *name* is specified as the errors parameter.

For encoding, *error_handler* will be called with a `UnicodeEncodeError` instance, which contains information about the location of the error. The error handler must either raise this or a different exception, or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The replacement may be either *str* or *bytes*. If the replacement is bytes, the encoder will simply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement. Encoding continues on original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an `IndexError` will be raised.

Decoding and translating works similarly, except `UnicodeDecodeError` or `UnicodeTranslateError` will be passed to the handler and that the replacement from the error handler will be put into the output directly.

Previously registered error handlers (including the standard error handlers) can be looked up by name :

`codecs.lookup_error` (*name*)

Return the error handler previously registered under the name *name*.

Raises a `LookupError` in case the handler cannot be found.

The following standard error handlers are also made available as module level functions :

`codecs.strict_errors` (*exception*)

Implements the 'strict' error handling : each encoding or decoding error raises a `UnicodeError`.

`codecs.replace_errors` (*exception*)

Implements the 'replace' error handling (for *text encodings* only) : substitutes ' ? ' for encoding errors (to be encoded by the codec), and '\ufffd' (the Unicode replacement character) for decoding errors.

`codecs.ignore_errors` (*exception*)

Implements the 'ignore' error handling : malformed data is ignored and encoding or decoding is continued without further notice.

`codecs.xmlcharrefreplace_errors` (*exception*)

Implements the 'xmlcharrefreplace' error handling (for encoding with *text encodings* only) : the unencodable character is replaced by an appropriate XML character reference.

`codecs.backslashreplace_errors` (*exception*)

Implements the 'backslashreplace' error handling (for *text encodings* only) : malformed data is replaced by a backslashed escape sequence.

`codecs.namereplace_errors` (*exception*)

Implements the 'namereplace' error handling (for encoding with *text encodings* only) : the unencodable character is replaced by a `\N{...}` escape sequence.

Nouveau dans la version 3.5.

Stateless Encoding and Decoding

The base `Codec` class defines these methods which also define the function interfaces of the stateless encoder and decoder :

`Codec.encode` (*input* [, *errors*])

Encodes the object *input* and returns a tuple (output object, length consumed). For instance, *text encoding* converts a string object to a bytes object using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

The *errors* argument defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the `Codec` instance. Use `StreamWriter` for codecs which have to keep state in order to make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

`Codec.decode(input[, errors])`

Decodes the object *input* and returns a tuple (output object, length consumed). For instance, for a *text encoding*, decoding converts a bytes object encoded using a particular character set encoding to a string object.

For text encodings and bytes-to-bytes codecs, *input* must be a bytes object or one which provides the read-only buffer interface – for example, buffer objects and memory mapped files.

The *errors* argument defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the `Codec` instance. Use *StreamReader* for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

Incremental Encoding and Decoding

The *IncrementalEncoder* and *IncrementalDecoder* classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn't done with one call to the stateless encoder/decoder function, but with multiple calls to the *encode()/decode()* method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the *encode()/decode()* method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

IncrementalEncoder Objects

The *IncrementalEncoder* class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

class `codecs.IncrementalEncoder` (*errors*='strict')

Constructor for an *IncrementalEncoder* instance.

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *IncrementalEncoder* may implement different error handling schemes by providing the *errors* keyword argument. See *Gestionnaires d'erreurs* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the *IncrementalEncoder* object.

encode (*object*[, *final*])

Encodes *object* (taking the current state of the encoder into account) and returns the resulting encoded object.

If this is the last call to *encode()* *final* must be true (the default is false).

reset ()

Reset the encoder to the initial state. The output is discarded : call *encode(object, final=True)*, passing an empty byte or text string if necessary, to reset the encoder and to get the output.

getstate ()

Return the current state of the encoder which must be an integer. The implementation should make sure that 0 is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer).

setstate (*state*)

Set the state of the encoder to *state*. *state* must be an encoder state returned by *getstate()*.

IncrementalDecoder Objects

The `IncrementalDecoder` class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

class `codecs.IncrementalDecoder` (*errors*='strict')

Constructor for an `IncrementalDecoder` instance.

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalDecoder` may implement different error handling schemes by providing the *errors* keyword argument. See *Gestionnaires d'erreurs* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `IncrementalDecoder` object.

decode (*object*[, *final*])

Decodes *object* (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to `decode()` *final* must be true (the default is false). If *final* is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

reset ()

Reset the decoder to the initial state.

getstate ()

Return the current state of the decoder. This must be a tuple with two items, the first must be the buffer containing the still undecoded input. The second must be an integer and can be additional state info. (The implementation should make sure that 0 is the most common additional state info.) If this additional state info is 0 it must be possible to set the decoder to the state which has no input buffered and 0 as the additional state info, so that feeding the previously buffered input to the decoder returns it to the previous state without producing any output. (Additional state info that is more complicated than integers can be converted into an integer by marshaling/pickling the info and encoding the bytes of the resulting string into an integer.)

setstate (*state*)

Set the state of the encoder to *state*. *state* must be a decoder state returned by `getstate()`.

Stream Encoding and Decoding

The `StreamWriter` and `StreamReader` classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See `encodings.utf_8` for an example of how this is done.

StreamWriter Objects

The `StreamWriter` class is a subclass of `Codec` and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

class `codecs.StreamWriter` (*stream*, *errors*='strict')

Constructor for a `StreamWriter` instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for writing text or binary data, as appropriate for the specific codec.

The `StreamWriter` may implement different error handling schemes by providing the *errors* keyword argument. See *Gestionnaires d'erreurs* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamWriter` object.

write (*object*)

Writes the object's contents encoded to the stream.

writelines (*list*)

Writes the concatenated list of strings to the stream (possibly by reusing the `write()` method). The standard bytes-to-bytes codecs do not support this method.

reset ()

Flushes and resets the codec buffers used for keeping state.

Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the `StreamWriter` must also inherit all other methods and attributes from the underlying stream.

StreamReader Objects

The `StreamReader` class is a subclass of `Codec` and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

class `codecs.StreamReader` (*stream*, *errors*=*'strict'*)

Constructor for a `StreamReader` instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for reading text or binary data, as appropriate for the specific codec.

The `StreamReader` may implement different error handling schemes by providing the *errors* keyword argument. See *Gestionnaires d'erreurs* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamReader` object.

The set of allowed values for the *errors* argument can be extended with `register_error()`.

read ([*size*[, *chars*[, *firstline*]]])

Decodes data from the stream and returns the resulting object.

The *chars* argument indicates the number of decoded code points or bytes to return. The `read()` method will never return more data than requested, but it might return less, if there is not enough available.

The *size* argument indicates the approximate maximum number of encoded bytes or code points to read for decoding. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. This parameter is intended to prevent having to decode huge files in one step.

The *firstline* flag indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

readline ([*size*[, *keepends*]])

Read one line from the input stream and return the decoded data.

size, if given, is passed as size argument to the stream's `read()` method.

If *keepends* is false line-endings will be stripped from the lines returned.

readlines ([*sizehint*[, *keepends*]])

Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's decoder method and are included in the list entries if *keepends* is true.

sizehint, if given, is passed as the *size* argument to the stream's `read()` method.

reset ()

Resets the codec buffers used for keeping state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the *StreamReader* must also inherit all other methods and attributes from the underlying stream.

StreamReaderWriter Objects

The *StreamReaderWriter* is a convenience class that allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the *lookup()* function to construct the instance.

class `codecs.StreamReaderWriter` (*stream*, *Reader*, *Writer*, *errors*='strict')

Creates a *StreamReaderWriter* instance. *stream* must be a file-like object. *Reader* and *Writer* must be factory functions or classes providing the *StreamReader* and *StreamWriter* interface resp. Error handling is done in the same way as defined for the stream readers and writers.

StreamReaderWriter instances define the combined interfaces of *StreamReader* and *StreamWriter* classes. They inherit all other methods and attributes from the underlying stream.

StreamRecoder Objects

The *StreamRecoder* translates data from one encoding to another, which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the *lookup()* function to construct the instance.

class `codecs.StreamRecoder` (*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors*='strict')

Creates a *StreamRecoder* instance which implements a two-way conversion : *encode* and *decode* work on the frontend — the data visible to code calling *read()* and *write()*, while *Reader* and *Writer* work on the backend — the data in *stream*.

You can use these objects to do transparent transcodings from e.g. Latin-1 to UTF-8 and back.

The *stream* argument must be a file-like object.

The *encode* and *decode* arguments must adhere to the `Codec` interface. *Reader* and *Writer* must be factory functions or classes providing objects of the *StreamReader* and *StreamWriter* interface respectively.

Error handling is done in the same way as defined for the stream readers and writers.

StreamRecoder instances define the combined interfaces of *StreamReader* and *StreamWriter* classes. They inherit all other methods and attributes from the underlying stream.

7.2.2 Encodings and Unicode

Strings are stored internally as sequences of code points in range 0x0–0x10FFFF. (See [PEP 393](#) for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectively referred to as *text encodings*.

The simplest text encoding (called 'latin-1' or 'iso-8859-1') maps the code points 0–255 to the bytes 0x0–0xff, which means that a string object that contains code points above U+00FF can't be encoded with this codec. Doing so will raise a *UnicodeEncodeError* that looks like the following (although the details of the error message

may differ):UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256).

There's another group of encodings (the so called charmap encodings) that choose a different subset of all Unicode code points and how these code points are mapped to the bytes 0x0-0xff. To see how this is done simply open e.g. `encodings/cp1252.py` (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities : store the bytes in big endian or in little endian order. These two encodings are called UTF-32-BE and UTF-32-LE respectively. Their disadvantage is that if e.g. you use UTF-32-BE on a little endian machine you will always have to swap bytes on encoding and decoding. UTF-32 avoids this problem : bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a UTF-16 or UTF-32 byte sequence, there's the so called BOM (« Byte Order Mark »). This is the Unicode character U+FEFF. This character can be prepended to every UTF-16 or UTF-32 byte sequence. The byte swapped version of this character (0xFFFE) is an illegal character that may not appear in a Unicode text. So when the first character in an UTF-16 or UTF-32 byte sequence appears to be a U+FFFE the bytes have to be swapped on decoding. Unfortunately the character U+FEFF had a second purpose as a ZERO WIDTH NO-BREAK SPACE : a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using U+FEFF as a ZERO WIDTH NO-BREAK SPACE has been deprecated (with U+2060 (WORD JOINER) assuming this role). Nevertheless Unicode software still must be able to handle U+FEFF in both roles : as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a ZERO WIDTH NO-BREAK SPACE it's a normal character that will be decoded like any other.

There's another encoding that is able to encoding the full range of Unicode characters : UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts : marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four 1 bits followed by a 0 bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character) :

Range	Encoding
U-00000000 ... U-0000007F	0xxxxxxx
U-00000080 ... U-000007FF	110xxxxx 10xxxxxx
U-00000800 ... U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ... U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any U+FEFF character in the decoded string (even if it's the first character) is treated as a ZERO WIDTH NO-BREAK SPACE.

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python 2.5 calls "utf-8-sig") for its Notepad program : Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence : 0xef, 0xbb, 0xbf) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a utf-8-sig encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write 0xef, 0xbb, 0xbf as the

first three bytes to the file. On decoding `utf-8-sig` will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

7.2.3 Standard Encodings

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases; therefore, e.g. `'utf-8'` is a valid alias for the `'utf_8'` codec.

CPython implementation detail : Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of (case insensitive) aliases : `utf-8`, `utf8`, `latin-1`, `latin1`, `iso-8859-1`, `iso8859-1`, `mbcs` (Windows only), `ascii`, `us-ascii`, `utf-16`, `utf16`, `utf-32`, `utf32`, and the same using underscores instead of dashes. Using alternative aliases for these encodings may result in slower execution.

Modifié dans la version 3.6 : Optimization opportunity recognized for `us-ascii`.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist :

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

Codec	Aliases	Languages
<i>ascii</i>	<i>646, us-ascii</i>	Anglais
<i>big5</i>	<i>big5-tw, csbig5</i>	Chinois Traditionnel
<i>big5hkscs</i>	<i>big5-hkscs, hkscs</i>	Chinois Traditionnel
<i>cp037</i>	<i>IBM037, IBM039</i>	Anglais
<i>cp273</i>	<i>273, IBM273, csIBM273</i>	Allemand Nouveau dans la version 3.4.
<i>cp424</i>	<i>EBCDIC-CP-HE, IBM424</i>	Hébreux
<i>cp437</i>	<i>437, IBM437</i>	Anglais
<i>cp500</i>	<i>EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500</i>	Europe de l'ouest
<i>cp720</i>		Arabe
<i>cp737</i>		Grec
<i>cp775</i>	<i>IBM775</i>	Langues Baltiques
<i>cp850</i>	<i>850, IBM850</i>	Europe de l'ouest
<i>cp852</i>	<i>852, IBM852</i>	Europe centrale et Europe de l'Est
<i>cp855</i>	<i>855, IBM855</i>	Bulgare, Biélorusse, Macédonien, Russe, Serbe
<i>cp856</i>		Hébreux
<i>cp857</i>	<i>857, IBM857</i>	Turc
<i>cp858</i>	<i>858, IBM858</i>	Europe de l'ouest
<i>cp860</i>	<i>860, IBM860</i>	Portugais
<i>cp861</i>	<i>861, CP-IS, IBM861</i>	Islandais
<i>cp862</i>	<i>862, IBM862</i>	Hébreux
<i>cp863</i>	<i>863, IBM863</i>	Canadien
<i>cp864</i>	<i>IBM864</i>	Arabe

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Codec	Aliases	Languages
<i>cp865</i>	<i>865, IBM865</i>	Danish, Norwegian
<i>cp866</i>	<i>866, IBM866</i>	Russe
<i>cp869</i>	<i>869, CP-GR, IBM869</i>	Grec
<i>cp874</i>		Thai
<i>cp875</i>		Grec
<i>cp932</i>	<i>932, ms932, mskanji, ms-kanji</i>	Japanese
<i>cp949</i>	<i>949, ms949, uhc</i>	Korean
<i>cp950</i>	<i>950, ms950</i>	Chinois Traditionnel
<i>cp1006</i>		Urdu
<i>cp1026</i>	<i>ibm1026</i>	Turc
<i>cp1125</i>	<i>1125, ibm1125, cp866u, ruscii</i>	Ukrainian Nouveau dans la version 3.4.
<i>cp1140</i>	<i>ibm1140</i>	Europe de l'ouest
<i>cp1250</i>	<i>windows-1250</i>	Europe centrale et Europe de l'Est
<i>cp1251</i>	<i>windows-1251</i>	Bulgare, Biélorusse, Macédonien, Russe, Serbe
<i>cp1252</i>	<i>windows-1252</i>	Europe de l'ouest
<i>cp1253</i>	<i>windows-1253</i>	Grec
<i>cp1254</i>	<i>windows-1254</i>	Turc
<i>cp1255</i>	<i>windows-1255</i>	Hébreux
<i>cp1256</i>	<i>windows-1256</i>	Arabe
<i>cp1257</i>	<i>windows-1257</i>	Langues Baltiques
<i>cp1258</i>	<i>windows-1258</i>	Vietnamese
<i>cp65001</i>		Windows uniquement : Windows UTF-8 (CP_UTF8) Nouveau dans la version 3.3.
<i>euc_jp</i>	<i>eucjp, ujis, u-jis</i>	Japanese
<i>euc_jis_2004</i>	<i>jisx0213, eucjis2004</i>	Japanese
<i>euc_jisx0213</i>	<i>eucjisx0213</i>	Japanese
<i>euc_kr</i>	<i>euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001</i>	Korean
<i>gb2312</i>	<i>chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58</i>	Simplified Chinese
<i>gbk</i>	<i>936, cp936, ms936</i>	Unified Chinese
<i>gb18030</i>	<i>gb18030-2000</i>	Unified Chinese
<i>hz</i>	<i>hzgb, hz-gb, hz-gb-2312</i>	Simplified Chinese
<i>iso2022_jp</i>	<i>csiso2022jp, iso2022jp, iso-2022-jp</i>	Japanese
<i>iso2022_jp_1</i>	<i>iso2022jp-1, iso-2022-jp-1</i>	Japanese
<i>iso2022_jp_2</i>	<i>iso2022jp-2, iso-2022-jp-2</i>	Japanese, Korean, Simplified Chinese, Western Europe, Greek
<i>iso2022_jp_2004</i>	<i>iso2022jp-2004, iso-2022-jp-2004</i>	Japanese
<i>iso2022_jp_3</i>	<i>iso2022jp-3, iso-2022-jp-3</i>	Japanese
<i>iso2022_jp_ext</i>	<i>iso2022jp-ext, iso-2022-jp-ext</i>	Japanese
<i>iso2022_kr</i>	<i>csiso2022kr, iso2022kr, iso-2022-kr</i>	Korean
<i>latin_1</i>	<i>iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1</i>	Europe de l'Ouest
<i>iso8859_2</i>	<i>iso-8859-2, latin2, L2</i>	Europe centrale et Europe de l'Est
<i>iso8859_3</i>	<i>iso-8859-3, latin3, L3</i>	Esperanto, Maltese

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Codec	Aliases	Languages
<i>iso8859_4</i>	<i>iso-8859-4, latin4, L4</i>	Langues Baltiques
<i>iso8859_5</i>	<i>iso-8859-5, cyrillic</i>	Bulgare, Biélorusse, Macédonien, Russe, Serbe
<i>iso8859_6</i>	<i>iso-8859-6, arabic</i>	Arabe
<i>iso8859_7</i>	<i>iso-8859-7, greek, greek8</i>	Grec
<i>iso8859_8</i>	<i>iso-8859-8, hebrew</i>	Hébreux
<i>iso8859_9</i>	<i>iso-8859-9, latin5, L5</i>	Turc
<i>iso8859_10</i>	<i>iso-8859-10, latin6, L6</i>	Nordic languages
<i>iso8859_11</i>	<i>iso-8859-11, thai</i>	Thai languages
<i>iso8859_13</i>	<i>iso-8859-13, latin7, L7</i>	Langues Baltiques
<i>iso8859_14</i>	<i>iso-8859-14, latin8, L8</i>	Celtic languages
<i>iso8859_15</i>	<i>iso-8859-15, latin9, L9</i>	Europe de l'ouest
<i>iso8859_16</i>	<i>iso-8859-16, latin10, L10</i>	South-Eastern Europe
<i>johab</i>	<i>cp1361, ms1361</i>	Korean
<i>koi8_r</i>		Russe
<i>koi8_t</i>		<i>Tajik</i> Nouveau dans la version 3.5.
<i>koi8_u</i>		Ukrainian
<i>kz1048</i>	<i>kz_1048, strk1048_2002, rk1048</i>	Kazakh Nouveau dans la version 3.5.
<i>mac_cyrillic</i>	<i>maccyrillic</i>	Bulgare, Biélorusse, Macédonien, Russe, Serbe
<i>mac_greek</i>	<i>macgreek</i>	Grec
<i>mac_iceland</i>	<i>maciceland</i>	Islandais
<i>mac_latin2</i>	<i>maclatin2, maccentraleurope</i>	Europe centrale et Europe de l'Est
<i>mac_roman</i>	<i>macroman, macintosh</i>	Europe de l'ouest
<i>mac_turkish</i>	<i>macturkish</i>	Turc
<i>ptcp154</i>	<i>csptcp154, pt154, cp154, cyrillic-asian</i>	Kazakh
<i>shift_jis</i>	<i>csshiftjis, shiftjis, sjis, s_jis</i>	Japanese
<i>shift_jis_2004</i>	<i>shiftjis2004, sjis_2004, sjis2004</i>	Japanese
<i>shift_jisx0213</i>	<i>shiftjisx0213, sjisx0213, s_jisx0213</i>	Japanese
<i>utf_32</i>	<i>U32, utf32</i>	all languages
<i>utf_32_be</i>	<i>UTF-32BE</i>	all languages
<i>utf_32_le</i>	<i>UTF-32LE</i>	all languages
<i>utf_16</i>	<i>U16, utf16</i>	all languages
<i>utf_16_be</i>	<i>UTF-16BE</i>	all languages
<i>utf_16_le</i>	<i>UTF-16LE</i>	all languages
<i>utf_7</i>	<i>U7, unicode-1-1-utf-7</i>	all languages
<i>utf_8</i>	<i>U8, UTF, utf8</i>	all languages
<i>utf_8_sig</i>		all languages

Modifié dans la version 3.4 : The utf-16* and utf-32* encoders no longer allow surrogate code points (U+D800–U+DFFF) to be encoded. The utf-32* decoders no longer decode byte sequences that correspond to surrogate code points.

7.2.4 Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated purpose describes the encoding direction.

Text Encodings

The following codecs provide *str* to *bytes* encoding and *bytes-like object* to *str* decoding, similar to the Unicode text encodings.

Codec	Aliases	Objectif
idna		Implements RFC 3490 , see also <code>encodings.idna</code> . Only <code>errors='strict'</code> is supported.
mbscs	ansi, dbcs	Windows only : Encode operand according to the ANSI codepage (CP_ACP)
oem		Windows only : Encode operand according to the OEM codepage (CP_OEMCP) Nouveau dans la version 3.6.
palmos		Encoding of PalmOS 3.5
punycode		Implements RFC 3492 . Stateful codecs are not supported.
raw_unicode_escape		Latin-1 encoding with <code>\uXXXX</code> and <code>\UXXXXXXXX</code> for other code points. Existing backslashes are not escaped in any way. It is used in the Python pickle protocol.
undefined		Raise an exception for all conversions, even empty strings. The error handler is ignored.
unicode_escape		Encoding suitable as the contents of a Unicode literal in ASCII-encoded Python source code, except that quotes are not escaped. Decodes from Latin-1 source code. Beware that Python source code actually uses UTF-8 by default.
unicode_internal		Return the internal representation of the operand. Stateful codecs are not supported. Obsolète depuis la version 3.3 : This representation is obsoleted by PEP 393 .

Binary Transforms

The following codecs provide binary transforms : *bytes-like object* to *bytes* mappings. They are not supported by *bytes.decode()* (which only produces *str* output).

Codec	Aliases	Objectif	Encoder / decoder
base64_codec ¹	base64, base_64	Convert operand to multiline MIME base64 (the result always includes a trailing '\n') Modifié dans la version 3.4 : accepts any <i>bytes-like object</i> as input for encoding and decoding	<i>base64.encodebytes()</i> / <i>base64.decodebytes()</i>
bz2_codec	bz2	Compress the operand using bz2	<i>bz2.compress()</i> / <i>bz2.decompress()</i>
hex_codec	hex	Convert operand to hexadecimal representation, with two digits per byte	<i>binascii.b2a_hex()</i> / <i>binascii.a2b_hex()</i>
quopri_codec	quopri, quotedprintable, quoted_printable	Convert operand to MIME quoted printable	<i>quopri.encode()</i> with quotetabs=True / <i>quopri.decode()</i>
uu_codec	uu	Convert the operand using uuencode	<i>uu.encode()</i> / <i>uu.decode()</i>
zlib_codec	zip, zlib	Compress the operand using gzip	<i>zlib.compress()</i> / <i>zlib.decompress()</i>

Nouveau dans la version 3.2 : Restoration of the binary transforms.

Modifié dans la version 3.4 : Restoration of the aliases for the binary transforms.

Text Transforms

The following codec provides a text transform : a *str* to *str* mapping. It is not supported by *str.encode()* (which only produces *bytes* output).

Codec	Aliases	Objectif
rot_13	rot13	Returns the Caesar-cypher encryption of the operand

Nouveau dans la version 3.2 : Restoration of the `rot_13` text transform.

Modifié dans la version 3.4 : Restoration of the `rot13` alias.

1. In addition to *bytes-like objects*, 'base64_codec' also accepts ASCII-only instances of *str* for decoding

7.2.5 `encodings.idna` — Internationalized Domain Names in Applications

This module implements [RFC 3490](#) (Internationalized Domain Names in Applications) and [RFC 3492](#) (Nameprep : A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the `punycode` encoding and `stringprep`.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as `www.Alliancefrançaise.nu`) is converted into an ASCII-compatible encoding (ACE, such as `www.xn--alliancefranaise-npb.nu`). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP *Host* fields, and so on. This conversion is carried out in the application; if possible invisible to the user : The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

Python supports this conversion in several ways : the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in [section 3.1 of RFC 3490](#) and converting each label to ACE as required, and conversely separating an input byte string into labels based on the `.` separator and converting any ACE labels found into unicode. Furthermore, the `socket` module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the `socket` module. On top of that, modules that have host names as function parameters, such as `http.client` and `ftplib`, accept Unicode host names (`http.client` then also transparently sends an IDNA hostname in the *Host* field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed : Applications wishing to present such host names to the user should decode them to Unicode.

The module `encodings.idna` also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

`encodings.idna.nameprep(label)`

Return the nameprepped version of *label*. The implementation currently assumes query strings, so `AllowUnassigned` is `true`.

`encodings.idna.ToASCII(label)`

Convert a label to ASCII, as specified in [RFC 3490](#). `UseSTD3ASCIIRules` is assumed to be `false`.

`encodings.idna.ToUnicode(label)`

Convert a label to Unicode, as specified in [RFC 3490](#).

7.2.6 `encodings.mbc`s — Windows ANSI codepage

Encode operand according to the ANSI codepage (`CP_ACP`).

Availability : Windows only.

Modifié dans la version 3.3 : Support any error handler.

Modifié dans la version 3.2 : Before 3.2, the *errors* argument was ignored; `'replace'` was always used to encode, and `'ignore'` to decode.

7.2.7 `encodings.utf_8_sig` — UTF-8 codec with BOM signature

This module implements a variant of the UTF-8 codec : On encoding a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). For decoding an optional UTF-8 encoded BOM at the start of the data will be skipped.

Types de données

Les modules documentés dans ce chapitre fournissent une gamme de types de données spécialisés tel que les dates et les heures, les listes à type prédéfini, les *heap queue*, les queues synchronisées, et les ensembles.

Python fournit aussi quelques types natifs, typiquement *dict*, *list*, *set*, *frozenset*, et *tuple*. La classe *str* est utilisée pour stocker des chaînes Unicode, et la classe *bytes* des données binaires.

Les modules suivants sont documentés dans ce chapitre :

8.1 *datetime* — Types de base pour la date et l’heure

Code source : [Lib/datetime.py](#)

Le module *datetime* fournit des classes pour manipuler de façon simple ou plus complexe des dates et des heures. Bien que les calculs de date et d’heure sont gérés, l’implémentation est essentiellement tournée vers l’efficacité pour extraire des attributs pour les manipuler et les formater pour l’affichage. Pour d’autres fonctionnalités associées, voir aussi les modules *time* et *calendar*.

Il y a deux sortes d’objets *date* et *time* : les « naïfs » et les « avisés ».

Un objet avisé possède suffisamment de connaissance des règles à appliquer et des politiques d’ajustement de l’heure comme les informations sur les fuseaux horaires et l’heure d’été pour se situer de façon relative par rapport à d’autres objets avisés. Un objet avisé est utilisé pour représenté un moment précis de l’histoire qui n’est pas ouvert à l’interprétation¹.

Un objet naïf ne comporte pas assez d’informations pour se situer sans ambiguïté par rapport à d’autres objets *date/time*. Le fait qu’un objet naïf représente un Temps universel coordonné (UTC), une heure locale ou une heure dans un autre fuseau horaire dépend complètement du programme, tout comme un nombre peut représenter une longueur, un poids ou une distance pour le programme. Les objets naïfs sont simples à comprendre et il est aisé de travailler avec, au prix de négliger certains aspects de la réalité.

Pour les applications qui nécessitent des objets avisés, les objets *datetime* et *time* ont un attribut optionnel d’information sur le fuseau horaire, *tzinfo*, qui peut être réglé sur une instance d’une sous-classe de la classe abstraite *tzinfo*.

1. Si on ignore les effets de la Relativité

Ces objets `tzinfo` capturent l'information à propos du décalage avec le temps UTC, le nom du fuseau horaire, et si l'heure d'été est effective. Notez qu'une seule classe concrète `tzinfo`, la classe `timezone`, est proposée par le module `datetime`. La classe `timezone` représente des fuseaux horaires simples avec un décalage fixe par rapport à UTC, comme UTC lui-même ou les fuseaux EST et EDT d'Amérique du Nord. Gérer des fuseaux horaires d'un niveau de détails plus avancé est à la charge de l'application. Les règles d'ajustement du temps à travers le monde sont plus politiques que rationnelles, changent fréquemment, et il n'y a pas de standard qui vaille pour toute application, en dehors d'UTC.

Le module `datetime` exporte les constantes suivantes :

`datetime.MINYEAR`

Le numéro d'année le plus petit autorisé dans un objet `date` ou `datetime`. `MINYEAR` vaut 1.

`datetime.MAXYEAR`

Le numéro d'année le plus grand autorisé dans un objet `date` ou `datetime`. `MAXYEAR` vaut 9999.

Voir aussi :

Module `calendar` Fonctions génériques associées au calendrier.

Module `time` Accès aux données d'horaires et aux conversions associées.

8.1.1 Types disponibles

class `datetime.date`

Une date naïve idéalisée, en supposant que le calendrier Grégorien actuel a toujours existé et qu'il existera toujours.

Attributs : `year`, `month` et `day`.

class `datetime.time`

Un temps idéalisé, indépendant d'une date particulière, en supposant qu'une journée est composée d'exactly 24*60*60 secondes (il n'y a pas ici de notion de « seconde bissextile »). Attributs : `hour`, `minute`, `second`, `microsecond` et `tzinfo`.

class `datetime.datetime`

Une combinaison d'une date et d'une heure. Attributs : `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, et `tzinfo`.

class `datetime.timedelta`

Une durée qui exprime la différence entre deux instances de `date`, `time` ou `datetime` en microsecondes.

class `datetime.tzinfo`

Une classe de base abstraite pour les objets portant des informations sur les fuseaux horaires. Ceux-ci sont utilisés par les classes `datetime` et `time` pour donner une notion personnalisable d'ajustement d'heure (par exemple la prise en compte d'un fuseau horaire et/ou de l'heure d'été).

class `datetime.timezone`

Une classe qui implémente la classe de base abstraite `tzinfo` en tant qu'offset fixe par rapport au temps UTC.

Nouveau dans la version 3.2.

Les objets issus de ces types sont immuables.

Les objets de type `date` sont toujours naïfs.

Un objet de type `time` ou `datetime` peut être naïf ou avisé. Un objet `datetime d` est avisé si `d.tzinfo` ne vaut pas `None` et que `d.tzinfo.utcoffset(d)` ne renvoie pas `None`. Si `d.tzinfo` vaut `None` ou que `d.tzinfo` ne vaut pas `None` mais que `d.tzinfo.utcoffset(d)` renvoie `None`, alors `d` est naïf. Un objet `time t` est avisé si `t.tzinfo` ne vaut pas `None` et que `t.tzinfo.utcoffset(None)` ne renvoie pas `None`. Sinon, `t` est naïf.

La distinction entre naïf et avisé ne s'applique pas aux objets de type `timedelta`.

Relations entre les sous-classes :


```

object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime

```

8.1.2 Objets `timedelta`

Un objet `timedelta` représente une durée, c'est-à-dire la différence entre deux dates ou heures.

class `datetime.timedelta` (*days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0*)

Tous les paramètres sont optionnels et ont 0 comme valeur par défaut. Les paramètres peuvent être des entiers ou des flottants et ils peuvent être positifs ou négatifs.

Seuls les *jours*, les *secondes* et les *microsecondes* sont stockés en interne. Tous les paramètres sont convertis dans ces unités :

- Une milliseconde est convertie en 1000 microsecondes.
- Une minute est convertie en 60 secondes.
- Une heure est convertie en 3600 secondes.
- Une semaine est convertie en 7 jours.

et ensuite les jours, secondes et microsecondes sont normalisés pour que la représentation soit unique avec

- $0 \leq \text{microseconds} < 1000000$
- $0 \leq \text{secondes} < 3600 \times 24$ (le nombre de secondes dans une journée)
- $-999999999 \leq \text{days} \leq 999999999$

Si l'un des arguments est un flottant et qu'il y a des microsecondes décimales, les microsecondes décimales laissées par les arguments sont combinées et leur somme est arrondie à la microseconde la plus proche (en arrondissant les demis vers le nombre pair). Si aucun argument n'est flottant, les processus de conversion et de normalisation seront exacts (pas d'informations perdues).

Si la valeur normalisée des jours déborde de l'intervalle indiqué, une `OverflowError` est levée.

Notez que la normalisation de valeurs négatives peut être surprenante au premier abord. Par exemple,

```

>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)

```

Les attributs de la classe sont :

`timedelta.min`

L'objet `timedelta` le plus négatif, `timedelta(-999999999)`.

`timedelta.max`

L'objet `timedelta` le plus positif, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

`timedelta.resolution`

La plus petite différence entre des objets `timedelta` non égaux, `timedelta(microseconds=1)`.

Il est à noter, du fait de la normalisation, que `timedelta.max > -timedelta.min`. `-timedelta.max` n'est pas représentable sous la forme d'un objet `timedelta`.

Attributs de l'instance (en lecture seule) :

Attribut	Valeur
days	Entre -999999999 et 999999999 inclus
seconds	Entre 0 et 86399 inclus
microseconds	Entre 0 et 999999 inclus

Opérations gérées :

Opération	Résultat
<code>t1 = t2 + t3</code>	Somme de <i>t2</i> et <i>t3</i> . Ensuite <code>t1 - t2 == t3</code> et <code>t1 - t3 == t2</code> sont des expressions vraies. (1)
<code>t1 = t2 - t3</code>	Difference of <i>t2</i> and <i>t3</i> . Afterwards <code>t1 == t2 - t3</code> and <code>t2 == t1 + t3</code> are true. (1)(6)
<code>t1 = t2 * i</code> or <code>t1 = i * t2</code>	Delta multiplié par un entier. Ensuite <code>t1 // i == t2</code> est vrai, en admettant que <code>i != 0</code> .
	De manière générale, <code>t1 * i == t1 * (i-1) + t1</code> est vrai. (1)
<code>t1 = t2 * f</code> or <code>t1 = f * t2</code>	Delta multiplié par un flottant. Le résultat est arrondi au multiple le plus proche de <code>timedelta.resolution</code> en utilisant la règle de l'arrondi au pair le plus proche.
<code>f = t2 / t3</code>	Division (3) de <i>t2</i> par <i>t3</i> . Renvoie un objet <i>float</i> .
<code>t1 = t2 / f</code> or <code>t1 = t2 / i</code>	Delta divisé par un flottant ou un entier. Le résultat est arrondi au multiple le plus proche de <code>timedelta.resolution</code> en utilisant la règle de l'arrondi au pair le plus proche.
<code>t1 = t2 // i</code> or <code>t1 = t2 // t3</code>	Le quotient est calculé et le reste (s'il y en a un) est ignoré. Dans le second cas, un entier est renvoyé. (3)
<code>t1 = t2 % t3</code>	Le reste est calculé comme un objet de type <i>timedelta</i> . (3)
<code>q, r = divmod(t1, t2)</code>	Calcule le quotient et le reste : <code>q = t1 // t2</code> (3) et <code>r = t1 % t2</code> . <i>q</i> est un entier et <i>r</i> est un objet <i>timedelta</i> .
<code>+t1</code>	Renvoie un objet <i>timedelta</i> avec la même valeur. (2)
<code>-t1</code>	équivalent à <code>timedelta(-t1.days, -t1.seconds, -t1.microseconds)</code> , et à <code>t1 * -1</code> . (1)(4)
<code>abs(t)</code>	équivalent à <code>+t</code> quand <code>t.days >= 0</code> , et à <code>-t</code> quand <code>t.days < 0</code> . (2)
<code>str(t)</code>	Renvoie une chaîne de la forme <code>[D] day[s], [H]H:MM:SS[.UUUUUU]</code> , où <i>D</i> est négatif pour <i>t</i> négatif. (5)
<code>repr(t)</code>	Renvoie une chaîne de la forme <code>datetime.timedelta(D[, S[, U]])</code> , où <i>D</i> est négatif pour <i>t</i> négatif. (5)

Notes :

- (1) Ceci est exact, mais peut provoquer un débordement.
- (2) Ceci est exact, et ne peut pas provoquer un débordement.
- (3) Une division par 0 provoque *ZeroDivisionError*.
- (4) `-timedelta.max` n'est pas représentable avec un objet *timedelta*.
- (5) La représentation en chaîne de caractères des objets *timedelta* est normalisée similairement à leur représentation interne. Cela amène à des résultats inhabituels pour des *timedeltas* négatifs. Par exemple :

```
>>> timedelta(hours=-5)
datetime.timedelta(-1, 68400)
>>> print(_)
-1 day, 19:00:00
```

- (6) The expression `t2 - t3` will always be equal to the expression `t2 + (-t3)` except when *t3* is equal to `timedelta.max`; in that case the former will produce a result while the latter will overflow.

En plus des opérations listées ci-dessus, les objets *timedelta* implémentent certaines additions et soustractions avec des objets *date* et *datetime* (voir ci-dessous).

Modifié dans la version 3.2 : La division entière et la vraie division d'un objet `timedelta` par un autre `timedelta` sont maintenant gérées, comme le sont les opérations de reste euclidien et la fonction `divmod()`. La vraie division et la multiplication d'un objet `timedelta` par un `float` sont maintenant implémentées.

Les comparaisons entre objets `timedelta` sont maintenant gérées avec le `timedelta` représentant la plus courte durée considéré comme le plus petit. Afin d'empêcher les comparaisons de types mixtes de retomber sur la comparaison par défaut par l'adresse de l'objet, quand un objet `timedelta` est comparé à un objet de type différent, une `TypeError` est levée à moins que la comparaison soit `==` ou `!=`. Ces derniers cas renvoient respectivement `False` et `True`.

Les objets `timedelta` sont *hashable* (utilisables comme clés de dictionnaires), implémentent le protocole *pickle* et, dans un contexte booléen, un `timedelta` est considéré vrai si et seulement si il n'est pas égal à `timedelta(0)`.

Méthodes de l'instance :

`timedelta.total_seconds()`

Renvoie le nombre total de secondes contenues dans la durée. Équivalent à `td / timedelta(seconds=1)`.

Notez que pour des intervalles de temps très larges (supérieurs à 270 ans sur la plupart des plateformes), cette méthode perdra la précision des microsecondes.

Nouveau dans la version 3.2.

Exemple d'utilisation :

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                           minutes=50, seconds=600) # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(3285), 9)
>>> three_years = nine_years // 3;
>>> three_years, three_years.days // 365
(datetime.timedelta(1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True
```

8.1.3 Objets date

Un objet `date` représente une date (année, mois et jour) dans un calendrier idéal, l'actuel calendrier grégorien étendu indéfiniment dans les deux directions. Le 1er janvier de l'an 1 est appelé le jour numéro 1, le 2 janvier de l'an 1 est appelé le jour numéro 2, et ainsi de suite. Cela correspond à la définition du calendrier « grégorien proleptique » dans le livre *Calendrical Calculations* de Dershowitz et Reingold, où il est la base de tous les calculs. Référez-vous au livre pour les algorithmes de conversion entre calendriers grégorien proleptique et les autres systèmes.

class `datetime.date` (*year, month, day*)

Tous les arguments sont requis. Les arguments peuvent être des entiers, dans les intervalles suivant :

— `MINYEAR <= year <= MAXYEAR`

— `1 <= month <= 12`

— `1 <= day <= nombre de jours dans le mois et l'année donnés`

Si un argument est donné en dehors de ces intervalles, une `ValueError` est levée.

Autres constructeurs, méthodes de classe :

classmethod `date.today()`

Renvoie la date locale courante. Cela est équivalent à `date.fromtimestamp(time.time())`.

classmethod `date.fromtimestamp(timestamp)`

Renvoie la date locale correspondant à l'horodatage (*timestamp* en anglais) *POSIX*, tel que renvoyé par `time.time()`. Elle peut lever une *OverflowError*, si l'horodatage est en dehors des bornes gérées par la fonction C `localtime()` de la plateforme, et une *OSError* en cas d'échec de `localtime()`. Il est commun d'être restreint aux années entre 1970 et 2038. Notez que sur les systèmes non *POSIX* qui incluent les secondes de décalage dans leur notion d'horodatage, ces secondes sont ignorées par `fromtimestamp()`.

Modifié dans la version 3.3 : Lève une *OverflowError* plutôt qu'une *ValueError* si l'horodatage (*timestamp* en anglais) est en dehors des bornes gérées par la fonction C `localtime()` de la plateforme. Lève une *OSError* plutôt qu'une *ValueError* en cas d'échec de `localtime()`.

classmethod `date.fromordinal(ordinal)`

Renvoie la date correspondant à l'ordinal grégorien proleptique, où le 1er janvier de l'an 1 a l'ordinal 1. *ValueError* est levée à moins que $1 \leq \text{ordinal} \leq \text{date.max.toordinal}()$. Pour toute date *d*, `date.fromordinal(d.toordinal()) == d`.

Attributs de la classe :

`date.min`

La plus vieille date représentable, `date(MINYEAR, 1, 1)`.

`date.max`

La dernière date représentable, `date(MAXYEAR, 12, 31)`.

`date.resolution`

La plus petite différence possible entre deux objets dates non-égaux, `timedelta(days=1)`.

Attributs de l'instance (en lecture seule) :

`date.year`

Entre *MINYEAR* et *MAXYEAR* inclus.

`date.month`

Entre 1 et 12 inclus.

`date.day`

Entre 1 et le nombre de jours du mois donné de l'année donnée.

Opérations gérées :

Opération	Résultat
<code>date2 = date1 + timedelta</code>	<i>date2</i> est décalée de <code>timedelta.days</code> jours par rapport à <i>date1</i> . (1)
<code>date2 = date1 - timedelta</code>	Calcule <i>date2</i> de façon à avoir <code>date2 + timedelta == date1</code> . (2)
<code>timedelta = date1 - date2</code>	(3)
<code>date1 < date2</code>	<i>date1</i> est considérée comme inférieure à <i>date2</i> quand <i>date1</i> précède <i>date2</i> dans le temps. (4)

Notes :

- (1) *date2* est déplacée en avant dans le temps si `timedelta.days > 0`, ou en arrière si `timedelta.days < 0`. Après quoi `date2 - date1 == timedelta.days * timedelta.seconds` et `timedelta.microseconds` sont ignorés. Une *OverflowError* est levée si `date2.year` devait être inférieure à *MINYEAR* ou supérieure à *MAXYEAR*.

- (2) `timedelta.seconds` and `timedelta.microseconds` are ignored.

- (3) Cela est exact, et ne peut pas dépasser les bornes. `timedelta.seconds` et `timedelta.microseconds` valent 0, et `date2 + timedelta == date1` après cela.
- (4) In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`. Date comparison raises `TypeError` if the other comparand isn't also a `date` object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a `date` object is compared to an object of a different type, `TypeError` is raised unless the comparison is `==` or `!=`. The latter cases return `False` or `True`, respectively.

Les dates peuvent être utilisées en tant que clés de dictionnaires. Dans un contexte booléen, tous les objets `date` sont considérés comme vrais.

Méthodes de l'instance :

`date.replace(year=self.year, month=self.month, day=self.day)`

Renvoie une date avec la même valeur, excepté pour les valeurs spécifiées par arguments nommés. Par exemple, si `d == date(2002, 12, 31)`, alors `d.replace(day=26) == date(2002, 12, 26)`.

`date.timetuple()`

Renvoie une `time.struct_time` telle que renvoyée par `time.localtime()`. Les heures, minutes et secondes valent 0, et le *flag DST* (heure d'été) est -1. `d.timetuple()` est équivalent à `time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))`, où `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` est le numéro du jour dans l'année courante, commençant avec 1 pour le 1er janvier.

`date.toordinal()`

Renvoie l'ordinal grégorien proleptique de la date, où le 1er janvier de l'an 1 a l'ordinal 1. Pour tout objet `date d`, `date.fromordinal(d.toordinal()) == d`.

`date.weekday()`

Renvoie le jour de la semaine sous forme de nombre, où lundi vaut 0 et dimanche vaut 6. Par exemple, `date(2002, 12, 4).weekday() == 2`, un mercredi. Voir aussi `isoweekday()`.

`date.isoweekday()`

Renvoie le jour de la semaine sous forme de nombre, où lundi vaut 1 et dimanche vaut 7. Par exemple, `date(2002, 12, 4).isoweekday() == 3`, un mercredi. Voir aussi `weekday()`, `isocalendar()`.

`date.isocalendar()`

Renvoie un *tuple* de 3 éléments, (année ISO, numéro de semaine ISO, jour de la semaine ISO).

Le calendrier ISO est une variante largement utilisée du calendrier grégorien. Voir <https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm> pour une bonne explication.

Une année ISO est composée de 52 ou 53 semaines pleines, où chaque semaine débute un lundi et se termine un dimanche. La première semaine d'une année ISO est la première semaine calendaire (grégorienne) de l'année comportant un jeudi. Elle est appelée la semaine numéro 1, et l'année ISO de ce mercredi est la même que son année grégorienne.

Par exemple, l'année 2004 débute un jeudi, donc la première semaine de l'année ISO 2004 débute le lundi 29 décembre 2003 et se termine le dimanche 4 janvier 2004, ainsi `date(2003, 12, 29).isocalendar() == (2004, 1, 1)` et `date(2004, 1, 4).isocalendar() == (2004, 1, 7)`.

`date.isoformat()`

Renvoie une chaîne de caractères représentant la date au format ISO 8601, « YYYY-MM-DD ». Par exemple, `date(2002, 12, 4).isoformat() == '2002-12-04'`.

`date.__str__()`

Pour une date `d`, `str(d)` est équivalent à `d.isoformat()`.

`date.ctime()`

Renvoie une chaîne de caractères représentant la date, par exemple `date(2002, 12, 4).ctime()`

`== 'Wed Dec 4 00:00:00 2002'.d.ctime()` est équivalent à `time.ctime(time.mktime(d.timetuple()))` sur les plateformes où la fonction C native `ctime()` (que `time.ctime()` invoque, mais pas `date.ctime()`) est conforme au standard C.

`date.strftime(format)`

Renvoie une chaîne de caractères représentant la date, contrôlée par une chaîne de formatage explicite. Les codes de formatage se référant aux heures, minutes ou secondes auront pour valeur 0. Pour une liste complète des directives de formatage, voir *Comportement de strftime() et strptime()*.

`date.__format__(format)`

Identique à `date.strftime()`. Cela permet de spécifier une chaîne de formatage pour un objet `date` dans une chaîne de formatage littérale et à l'utilisation de `str.format()`. Pour une liste complète des directives de formatage, voir *Comportement de strftime() et strptime()*.

Exemple de décompte des jours avant un évènement :

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

Exemple d'utilisation de la classe `date` :

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002          # year
3             # month
11            # day
0
0
0
0             # weekday (0 = Monday)
70            # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002          # ISO year
11            # ISO week number
1             # ISO day number ( 1 = Monday )
>>> d.isoformat()
'2002-03-11'
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'
```

8.1.4 Objets `datetime`

Un objet `datetime` est un objet comportant toutes les informations d'un objet `date` et d'un objet `time`. Comme un objet `date`, un objet `datetime` utilise l'actuel calendrier Grégorien étendu vers le passé et le futur ; comme un objet `time`, un objet `datetime` suppose qu'il y a exactement 3600*24 secondes chaque jour.

Constructeur :

class `datetime.datetime` (*year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0*)

Les arguments *year, month* et *day* sont requis. *tzinfo* peut être `None`, ou une instance d'une sous-classe de `tzinfo`. Les arguments restant doivent être des nombres, dans les intervalles suivants :

- `MINYEAR <= year <= MAXYEAR`,
- `1 <= month <= 12`,
- `1 <= day <= nombre de jours dans le mois donné de l'année donnée`,
- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold` in `[0, 1]`.

Si un argument est donné en dehors de ces intervalles, une `valueError` est levée.

Nouveau dans la version 3.6 : Ajout de l'argument `fold`.

Autres constructeurs, méthodes de classe :

classmethod `datetime.today()`

Renvoie le `datetime` local courant, avec *tzinfo* à `None`. Cela est équivalent à `datetime.fromtimestamp(time.time())`. Voir aussi `now()`, `fromtimestamp()`.

classmethod `datetime.now` (*tz=None*)

Renvoie la date et l'heure courantes locales. Si l'argument optionnel *tz* est `None` ou n'est pas spécifié, la méthode est similaire à `today()`, mais, si possible, apporte plus de précisions que ce qui peut être trouvé à travers un horodatage `time.time()` (par exemple, cela peut être possible sur des plateformes fournissant la fonction `C gettimeofday()`).

Si *tz* n'est pas `None`, il doit être une instance d'une sous-classe `tzinfo`, et la date et l'heure courantes sont converties vers le fuseau horaire *tz*. Dans ce cas le résultat est équivalent à `tz.fromutc(datetime.utcnow())`. `replace(tzinfo=tz)`. Voir aussi `today()`, `utcnow()`.

classmethod `datetime.utcnow()`

Renvoie la date et l'heure UTC courantes, avec *tzinfo* à `None`. C'est semblable à `now()`, mais renvoie la date et l'heure UTC courantes, comme un objet `datetime` naïf. Un `datetime` UTC courant avisé peut être obtenu en appelant `datetime.now(timezone.utc)`. Voir aussi `now()`.

classmethod `datetime.fromtimestamp` (*timestamp, tz=None*)

Renvoie la date et l'heure locales correspondant à l'horodatage (*timestamp* en anglais) *POSIX*, comme renvoyé par `time.time()`. Si l'argument optionnel *tz* est `None` ou n'est pas spécifié, l'horodatage est converti vers la date et l'heure locales de la plateforme, et l'objet `datetime` renvoyé est naïf.

Si `tz` n'est pas `None`, il doit être une instance d'une sous-classe `tzinfo`, et l'horodatage (`timestamp` en anglais) est converti vers le fuseau horaire `tz`. Dans ce cas le résultat est équivalent à `tz.fromutc(datetime.utcnow().replace(tzinfo=tz))`.

`fromtimestamp()` peut lever une `OverflowError`, si l'horodatage est en dehors de l'intervalle de valeurs gérées par les fonctions C `localtime()` ou `gmtime()` de la plateforme, et une `OSError` en cas d'échec de `localtime()` ou `gmtime()`. Il est courant d'être restreint aux années de 1970 à 2038. Notez que sur les systèmes non *POSIX* qui incluent les secondes intercalaires dans leur notion d'horodatage, les secondes intercalaires sont ignorées par `fromtimestamp()`, et il est alors possible d'avoir deux horodatages différant d'une seconde produisant un objet `datetime` identique. Voir aussi `utcfromtimestamp()`.

Modifié dans la version 3.3 : Lève une `OverflowError` plutôt qu'une `ValueError` si l'horodatage est en dehors de l'intervalle de valeurs gérées par les fonctions C `localtime()` ou `gmtime()` de la plateforme. Lève une `OSError` plutôt qu'une `ValueError` en cas d'échec de `localtime()` ou `gmtime()`.

Modifié dans la version 3.6 : `fromtimestamp()` peut renvoyer des instances avec l'attribut `fold` à 1.

classmethod `datetime.utcnow(timestamp)`

Renvoie le `datetime` UTC correspondant à l'horodatage (`timestamp` en anglais) *POSIX*, avec `tzinfo` à `None`. Cela peut lever une `OverflowError`, si l'horodatage est en dehors de l'intervalle de valeurs gérées par la fonction C `gmtime()` de la plateforme, et une `OSError` en cas d'échec de `gmtime()`. Il est courant d'être restreint aux années de 1970 à 2038.

Pour obtenir un objet `datetime` avisé, appelez `fromtimestamp()` :

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

Sur les plateformes respectant *POSIX*, cela est équivalent à l'expression suivante :

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

excepté que la dernière formule gère l'intervalle complet des années entre *MINYEAR* et *MAXYEAR* incluses.

Modifié dans la version 3.3 : Lève une `OverflowError` plutôt qu'une `ValueError` si l'horodatage est en dehors de l'intervalle de valeurs gérées par la fonction C `gmtime()` de la plateforme. Lève une `OSError` plutôt qu'une `ValueError` en cas d'échec de `gmtime()`.

classmethod `datetime.fromordinal(ordinal)`

Renvoie le `datetime` correspondant à l'ordinal du calendrier grégorien proleptique, où le 1er janvier de l'an 1 a l'ordinal 1. Une `ValueError` est levée à moins que `1 <= ordinal <= datetime.max.toordinal()`. Les heures, minutes, secondes et microsecondes du résultat valent toutes 0, et `tzinfo` est `None`.

classmethod `datetime.combine(date, time, tzinfo=self.tzinfo)`

Renvoie un nouvel objet `datetime` dont les composants de date sont égaux à ceux de l'objet `date` donné, et donc les composants de temps sont égaux à ceux de l'objet `time` donné. Si l'argument `tzinfo` est fourni, sa valeur est utilisée pour initialiser l'attribut `tzinfo` du résultat, autrement l'attribut `tzinfo` de l'argument `time` est utilisé.

Pour tout objet `datetime d*`, "`d == datetime.combine(d.date(), d.time(), d.tzinfo)`". Si `*date` est un objet `datetime`, ses composants de temps et attributs `tzinfo` sont ignorés.

Modifié dans la version 3.6 : Ajout de l'argument `tzinfo`.

classmethod `datetime.strptime(date_string, format)`

Renvoie un `datetime` correspondant à la chaîne `date_string`, analysée conformément à `format`. Cela est équivalent à `datetime(*(time.strptime(date_string, format)[0:6]))`. Une `ValueError` est levée si `date_string` et `format` ne peuvent être analysées par `time.strptime()` ou si elle renvoie une valeur qui n'est pas un *tuple-temps*. Pour une liste complète des directives de formatage, voir *Comportement de strptime() et strftime()*.

Attributs de la classe :

`datetime.min`

Le plus ancien `datetime` représentable, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

`datetime.max`

Le dernier `datetime` représentable, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

`tzinfo=None`).

`datetime.resolution`

La plus petite différence possible entre deux objets `datetime` non-égaux, `timedelta(microseconds=1)`.

Attributs de l'instance (en lecture seule) :

`datetime.year`

Entre `MINYEAR` et `MAXYEAR` inclus.

`datetime.month`

Entre 1 et 12 inclus.

`datetime.day`

Entre 1 et le nombre de jours du mois donné de l'année donnée.

`datetime.hour`

Dans `range(24)`.

`datetime.minute`

Dans `range(60)`.

`datetime.second`

Dans `range(60)`.

`datetime.microsecond`

Dans `range(1000000)`.

`datetime.tzinfo`

L'objet passé en tant que paramètre `tzinfo` du constructeur de la classe `datetime` ou `None` si aucun n'a été donné.

`datetime.fold`

0 ou 1. Utilisé pour désambigüiser les heures dans un intervalle répété. (Un intervalle répété apparaît quand l'horloge est retardée à la fin de l'heure d'été ou quand le décalage UTC du fuseau courant et décrémente pour des raisons politiques.) La valeur 0 (1) représente le plus ancien (récent) des deux moments représentés par la même heure.

Nouveau dans la version 3.6.

Opérations gérées :

Opération	Résultat
<code>datetime2 = datetime1 + timedelta</code>	(1)
<code>datetime2 = datetime1 - timedelta</code>	(2)
<code>timedelta = datetime1 - datetime2</code>	(3)
<code>datetime1 < datetime2</code>	Compare <code>datetime</code> à <code>datetime</code> . (4)

- (1) `datetime2` est décalé d'une durée `timedelta` par rapport à `datetime1`, en avant dans le temps si `timedelta.days > 0`, ou en arrière si `timedelta.days < 0`. Le résultat a le même attribut `tzinfo` que le `datetime` d'entrée, et `datetime2 - datetime1 == timedelta` après l'opération. Une `OverflowError` est levée si `datetime2.year` devait être inférieure à `MINYEAR` ou supérieure à `MAXYEAR`. Notez qu'aucun ajustement de fuseau horaire n'est réalisé même si l'entrée est avisée.
- (2) Computes the `datetime2` such that `datetime2 + timedelta == datetime1`. As for addition, the result has the same `tzinfo` attribute as the input `datetime`, and no time zone adjustments are done even if the input is aware.
- (3) La soustraction d'un `datetime` à un autre `datetime` n'est définie que si les deux opérandes sont naïfs, ou s'ils sont les deux avisés. Si l'un est avisé et que l'autre est naïf, une `TypeError` est levée.
Si les deux sont naïfs, ou que les deux sont avisés et ont le même attribut `tzinfo`, les attributs `tzinfo` sont ignorés, et le résultat est un objet `timedelta` tel que `datetime2 + t == datetime1`. Aucun ajustement de fuseau horaire n'a lieu dans ce cas.

Si les deux sont avisés mais ont des attributs `tzinfo` différents, `a-b` agit comme si `a` et `b` étaient premièrement convertis vers des `datetimes` UTC naïfs. Le résultat est `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` à l'exception que l'implémentation ne produit jamais de débordement.

- (4) `datetime1` est considéré inférieur à `datetime2` quand il le précède dans le temps.

Si un opérande est naïf et l'autre avisé, une `TypeError` est levée si une comparaison d'ordre est attendue. Pour les comparaisons d'égalité, les instances naïves ne sont jamais égales aux instances avisées.

Si les deux opérandes sont avisés, et ont le même attribut `tzinfo`, l'attribut commun `tzinfo` est ignoré et les `datetimes` de base sont comparés. Si les deux opérandes sont avisés et ont des attributs `tzinfo` différents, les opérandes sont premièrement ajustés en soustrayant leurs décalages UTC (obtenus depuis `self.utcoffset()`). Modifié dans la version 3.3 : Les comparaisons d'égalité entre des instances `datetime` naïves et avisées ne lèvent pas de `TypeError`.

Note : Afin d'empêcher la comparaison de retomber sur le schéma par défaut de comparaison des adresses des objets, la comparaison `datetime` lève normalement une `TypeError` si l'autre opérande n'est pas aussi un objet `datetime`. Cependant, `NotImplemented` est renvoyé à la place si l'autre opérande a un attribut `timetuple()`. Cela permet à d'autres types d'objets dates d'implémenter la comparaison entre types mixtes. Sinon, quand un objet `datetime` est comparé à un objet d'un type différent, une `TypeError` est levée à moins que la comparaison soit `==` ou `!=`. Ces derniers cas renvoient respectivement `False` et `True`.

Les objets `datetime` peuvent être utilisés comme clés de dictionnaires. Dans les contextes booléens, tous les objets `datetime` sont considérés vrais.

Méthodes de l'instance :

`datetime.date()`

Renvoie un objet `date` avec les mêmes année, mois et jour.

`datetime.time()`

Renvoie un objet `time` avec les mêmes heure, minute, seconde, microseconde et `fold`. `tzinfo` est `None`. Voir aussi la méthode `timetz()`.

Modifié dans la version 3.6 : La valeur `fold` est copiée vers l'objet `time` renvoyé.

`datetime.timetz()`

Renvoie un objet `time` avec les mêmes attributs heure, minute, seconde, microseconde, `fold` et `tzinfo`. Voir aussi la méthode `time()`.

Modifié dans la version 3.6 : La valeur `fold` est copiée vers l'objet `time` renvoyé.

`datetime.replace(year=self.year, month=self.month, day=self.day, hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo, *fold=0)`

Renvoie un `datetime` avec les mêmes attributs, exceptés ceux dont de nouvelles valeurs sont données par les arguments nommés correspondant. Notez que `tzinfo=None` peut être spécifié pour créer un `datetime` naïf depuis un `datetime` avisé sans conversion de la date ou de l'heure.

Nouveau dans la version 3.6 : Ajout de l'argument `fold`.

`datetime.astimezone(tz=None)`

Renvoie un objet `datetime` avec un nouvel attribut `tzinfo` valant `tz`, ajustant la date et l'heure pour que le résultat soit le même temps UTC que `self`, mais dans le temps local au fuseau `tz`.

Si fourni, `tz` doit être une instance d'une sous-classe `tzinfo`, et ses méthodes `utcoffset()` et `dst()` ne doivent pas renvoyer `None`. Si `self` est naïf, Python considère que le temps est exprimé dans le fuseau horaire du système.

Si appelé sans arguments (ou si `tz=None`) le fuseau horaire local du système est utilisé comme fuseau horaire cible. L'attribut `.tzinfo` de l'instance `datetime` convertie aura pour valeur une instance de `timezone` avec le nom de fuseau et le décalage obtenus depuis l'OS.

Si `self.tzinfo` est `tz`, `self.astimezone(tz)` est égal à `self` : aucun ajustement de date ou d'heure n'est réalisé. Sinon le résultat est le temps local dans le fuseau `tz` représentant le même temps UTC que `self` : après `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` aura les mêmes données de date et d'heure que `dt - dt.utcoffset()`.

Si vous voulez seulement associer un fuseau horaire `tz` à un *datetime* `dt` sans ajustement des données de date et d'heure, utilisez `dt.replace(tzinfo=tz)`. Si vous voulez seulement supprimer le fuseau horaire d'un *datetime* `dt` avisé sans conversion des données de date et d'heure, utilisez `dt.replace(tzinfo=None)`.

Notez que la méthode par défaut `tzinfo.fromutc()` peut être redéfinie dans une sous-classe `tzinfo` pour affecter le résultat renvoyé par `astimezone()`. En ignorant les cas d'erreurs, `astimezone()` se comporte comme :

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

Modifié dans la version 3.3 : `tz` peut maintenant être omis.

Modifié dans la version 3.6 : La méthode `astimezone()` peut maintenant être appelée sur des instances naïves qui sont supposées représenter un temps local au système.

`datetime.utcoffset()`

Si `tzinfo` est `None`, renvoie `None`, sinon renvoie `self.tzinfo.utcoffset(self)`, et lève une exception si l'expression précédente ne renvoie pas `None` ou un objet `timedelta` représentant un nombre entier de minutes de magnitude inférieure à un jour.

`datetime.dst()`

Si `tzinfo` est `None`, renvoie `None`, sinon renvoie `self.tzinfo.dst(self)`, et lève une exception si l'expression précédente ne renvoie pas `None` ou un objet `timedelta` représentant un nombre entier de minutes de magnitude inférieure à un jour.

`datetime.tzname()`

Si `tzinfo` est `None`, renvoie `None`, sinon renvoie `self.tzinfo.tzname(self)`, lève une exception si l'expression précédente ne renvoie pas `None` ou une chaîne de caractères,

`datetime.timetuple()`

Renvoie un `time.struct_time` comme renvoyé par `time.localtime()`. `d.timetuple()` est équivalent à `time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst))`, où `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` est le numéro de jour dans l'année courante commençant avec 1 pour le 1er janvier. L'option `tm_isdst` du résultat est attribuée selon la méthode `dst()` : si `tzinfo` est `None` ou que `dst()` renvoie `None`, `tm_isdst` est mise à -1 ; sinon, si `dst()` renvoie une valeur non-nulle, `tm_isdst` est mise à 1 ; sinon `tm_isdst` est mise à 0.

`datetime.utctimetuple()`

Si l'instance de *datetime* `d` est naïve, cela est équivalent à `d.timetuple()`, excepté que `tm_isdst` est forcé à 0 sans tenir compte de ce que renvoie `d.dst()`. L'heure d'été n'est jamais effective pour un temps UTC. Si `d` est avisé, il est normalisé vers un temps UTC, en lui soustrayant `d.utcoffset()`, et un `time.struct_time` est renvoyé pour le temps normalisé. `tm_isdst` est forcé à 0. Notez qu'une `OverflowError` peut être levée si `d.year` vaut `MINYEAR` ou `MAXYEAR` et que l'ajustement UTC fait dépasser les bornes.

`datetime.toordinal()`

Renvoie l'ordinal du calendrier géorgien proleptique de cette date. Identique à `self.date().toordinal()`.

`datetime.timestamp()`

Renvoie l'horodatage (*timestamp* en anglais) *POSIX* correspondant à l'instance *datetime*. La valeur renvoyée est

un *float* similaire à ceux renvoyés par `time.time()`.

Les instances naïves de `datetime` sont supposées représenter un temps local et cette méthode se base sur la fonction C `mktime()` de la plateforme pour opérer la conversion. Comme `datetime` gère un intervalle de valeurs plus large que `mktime()` sur beaucoup de plateformes, cette méthode peut lever une `OverflowError` pour les temps trop éloignés dans le passé ou le futur.

Pour les instances `datetime` avisées, la valeur renvoyée est calculée comme suit :

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

Nouveau dans la version 3.3.

Modifié dans la version 3.6 : La méthode `timestamp()` utilise l'attribut `fold` pour désambiguïser le temps dans un intervalle répété.

Note : Il n'y a pas de méthode pour obtenir l'horodatage (*timestamp* en anglais) *POSIX* directement depuis une instance `datetime` naïve représentant un temps UTC. Si votre application utilise cette convention et que le fuseau horaire de votre système est UTC, vous pouvez obtenir l'horodatage *POSIX* en fournissant `tzinfo=timezone.utc` :

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

ou en calculant l'horodatage (*timestamp* en anglais) directement :

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

`datetime.weekday()`

Renvoie le jour de la semaine sous forme de nombre, où lundi vaut 0 et dimanche vaut 6. Identique à `self.date().weekday()`. Voir aussi `isoweekday()`.

`datetime.isoweekday()`

Renvoie le jour de la semaine sous forme de nombre, où lundi vaut 1 et dimanche vaut 7. Identique à `self.date().isoweekday()`. Voir aussi `weekday()`, `isocalendar()`.

`datetime.isocalendar()`

Renvoie un *tuple* de 3 éléments, (année ISO, numéro de semaine ISO, jour de la semaine ISO). Identique à `self.date().isocalendar()`.

`datetime.isoformat(sep='T', timespec='auto')`

Renvoie une chaîne représentant la date et l'heure au format ISO 8601, YYYY-MM-DDTHH:MM:SS.mmmmmmm ou, si *microsecond* vaut 0, YYYY-MM-DDTHH:MM:SS

Si `utcoffset()` ne renvoie pas `None`, une chaîne de 6 caractères est ajoutée, donnant le décalage UTC en heures et minutes (relatives) : YYYY-MM-DDTHH:MM:SS.mmmmmmm+HH:MM ou, si *microsecond* vaut 0, YYYY-MM-DDTHH:MM:SS+HH:MM

L'argument optionnel *sep* (valant par défaut 'T') est un séparateur d'un caractère, placé entre les portions du résultat correspondant à la date et à l'heure. Par exemple,

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

L'argument optionnel *timespec* spécifie le nombre de composants additionnels de temps à inclure (par défaut 'auto'). Il peut valoir l'une des valeurs suivantes :

- 'auto' : Identique à 'seconds' si *microsecond* vaut 0, à 'microseconds' sinon.
- 'hours' : Inclut *hour* au format à deux chiffres HH.

- 'minutes' : Inclut *hour* et *minute* au format HH :MM.
- 'seconds' : Inclut *hour*, *minute* et *second* au format HH :MM :SS.
- 'milliseconds' : Inclut le temps complet, mais tronque la partie fractionnaire des millisecondes, au format HH:MM:SS.sss.
- 'microseconds' : Inclut le temps complet, au format HH :MM :SS.mmmmmmm.

Note : Les composants de temps exclus sont tronqués et non arrondis.

Une *ValueError* sera levée en cas d'argument *timespec* invalide.

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

Nouveau dans la version 3.6 : Ajout de l'argument *timespec*.

`datetime.__str__()`

Pour une instance *d* de *datetime*, `str(d)` est équivalent à `d.isoformat(' ')`.

`datetime.ctime()`

Renvoie une chaîne représentant la date et l'heure, par exemple `datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'`. `d.ctime()` est équivalent à `time.ctime(time.mktime(d.timetuple()))` sur les plateformes où la fonction C native `ctime()` (invoquée par `time.ctime()` mais pas par `datetime.ctime()`) est conforme au standard C.

`datetime.strftime(format)`

Renvoie une chaîne représentant la date et l'heure, contrôlée par une chaîne de format explicite. Pour une liste complète des directives de formatage, voir *Comportement de strftime() et strptime()*.

`datetime.__format__(format)`

Identique à `datetime.strftime()`. Cela permet de spécifier une chaîne de format pour un objet *datetime* dans une chaîne de formatage littérale et en utilisant `str.format()`. Pour une liste complète des directives de formatage, voir *Comportement de strftime() et strptime()*.

Exemples d'utilisation des objets *datetime* :

```
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)    # GMT +1
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
```

(suite sur la page suivante)

(suite de la page précédente)

```

...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None
>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday
>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}'.format(dt, "day",
↪ "month", "time")
'The day is 21, the month is November, the time is 04:30PM.'

```

Utilisation de `datetime` avec `tzinfo` :

```

>>> from datetime import timedelta, datetime, tzinfo
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1) + self.dst(dt)
...     def dst(self, dt):
...         # DST starts last Sunday in March
...         d = datetime(dt.year, 4, 1) # ends last Sunday in October
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <= dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self, dt):
...         return "GMT +1"
...
>>> class GMT2(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=2) + self.dst(dt)
...     def dst(self, dt):
...         d = datetime(dt.year, 4, 1)
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <= dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self, dt):

```

(suite sur la page suivante)

(suite de la page précédente)

```

...         return "GMT +2"
...
>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()
datetime.timedelta(0)
>>> dt1.utcoffset()
datetime.timedelta(0, 3600)
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()
datetime.timedelta(0, 3600)
>>> dt2.utcoffset()
datetime.timedelta(0, 7200)
>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(GMT2())
>>> dt3
datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x...>)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=<GMT1 object at 0x...>)
>>> dt2.utctimetuple() == dt3.utctimetuple()
True

```

8.1.5 Objets `time`

Un objet `time` représente une heure (locale) du jour, indépendante de tout jour particulier, et sujette à des ajustements par un objet `tzinfo`.

class `datetime.time` (*hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0*)

Tous les arguments sont optionnels. `tzinfo` peut être `None` ou une instance d'une sous-classe `tzinfo`. Les autres arguments doivent être des nombres entiers, dans les intervalles suivants :

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold` in `[0, 1]`.

Si un argument est fourni en dehors de ces bornes, une `ValueError` est levée. Ils valent tous 0 par défaut, à l'exception de `tzinfo` qui vaut `None`.

Attributs de la classe :

`time.min`

Le plus petit objet `time` représentable, `time(0, 0, 0, 0)`.

`time.max`

Le plus grand objet `time` représentable, `time(23, 59, 59, 999999)`.

`time.resolution`

La plus petite différence possible entre deux objets `time` non-égaux, `timedelta(microseconds=1)`, notez cependant que les objets `time` n'implémentent pas d'opérations arithmétiques.

Attributs de l'instance (en lecture seule) :

`time.hour`

Dans `range(24)`.

`time.minute`

Dans `range(60)`.

`time.second`

Dans `range(60)`.

`time.microsecond`

Dans `range(1000000)`.

`time.tzinfo`

L'objet passé comme argument `tzinfo` au constructeur de `time`, ou `None` si aucune valeur n'a été passée.

`time.fold`

0 ou 1. Utilisé pour désambigüiser les heures dans un intervalle répété. (Un intervalle répété apparaît quand l'horloge est retardée à la fin de l'heure d'été ou quand le décalage UTC du fuseau courant et décrémente pour des raisons politiques.) La valeur 0 (1) représente le plus ancien (récent) des deux moments représentés par la même heure.

Nouveau dans la version 3.6.

Opérations gérées :

- comparaison d'un `time` avec un autre `time`, où `a` est considéré inférieur à `b` s'il le précède dans le temps. Si un opérande est naïf et l'autre avisé, et qu'une relation d'ordre est attendue, une `TypeError` est levée. Pour les égalités, les instances naïves ne sont jamais égales aux instances avisées.

Si les deux opérandes sont avisés, et ont le même attribut `tzinfo`, l'attribut commun `tzinfo` est ignoré et les temps de base sont comparés. Si les deux opérandes sont avisés et ont des attributs `tzinfo` différents, ils sont d'abord ajustés en leur soustrayant leurs décalages UTC (obtenus à l'aide de `self.utcoffset()`). Afin d'empêcher les comparaisons de types mixtes de retomber sur la comparaison par défaut par l'adresse de l'objet, quand un objet `time` est comparé à un objet de type différent, une `TypeError` est levée à moins que la comparaison soit `==` ou `!=`. Ces derniers cas renvoient respectivement `False` et `True`.

Modifié dans la version 3.3 : Les comparaisons d'égalité entre instances de `time` naïves et avisées ne lèvent pas de `TypeError`.

- hachage, utilisation comme clef de dictionnaire
- sérialisation (*pickling*) efficace

Dans un contexte booléen, un objet `time` est toujours considéré comme vrai.

Modifié dans la version 3.5 : Avant Python 3.5, un objet `time` était considéré comme faux s'il représentait minuit en UTC. Ce comportement était considéré comme obscur et propice aux erreurs, il a été supprimé en Python 3.5. Voir [bpo-13936](#) pour les détails complets.

Méthodes de l'instance :

`time.replace(hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo, *fold=0)`

Renvoie un objet `time` avec la même valeur, à l'exception des attributs dont une nouvelle valeur est spécifiée par les arguments nommés. Notez que `tzinfo=None` peut être spécifié pour créer une instance `time` naïve à partir d'une instance `time` avisée, sans conversion des données de temps.

Nouveau dans la version 3.6 : Ajout de l'argument `fold`.

`time.isoformat(timespec='auto')`

Renvoie une chaîne représentant l'heure au format ISO 8601, `HH:MM:SS.mmmmmmm` ou, si `microsecond` vaut 0, `HH:MM:SS`. Si `utcoffset()` ne renvoie pas `None`, une chaîne de 6 caractères est ajoutée, donnant le décalage UTC en heures et minutes (relatives) : `HH:MM:SS.mmmmmmm+HH:MM` ou, si `self.microsecond` vaut 0, `HH:MM:SS+HH:MM`.

L'argument optionnel `timespec` spécifie le nombre de composants additionnels de temps à inclure (par défaut `'auto'`). Il peut valoir l'une des valeurs suivantes :

- `'auto'` : Identique à `'seconds'` si `microsecond` vaut 0, à `'microseconds'` sinon.
- `'hours'` : Inclut `hour` au format à deux chiffres `HH`.
- `'minutes'` : Inclut `hour` et `minute` au format `HH:MM`.
- `'seconds'` : Inclut `hour`, `minute` et `second` au format `HH:MM:SS`.

- 'milliseconds' : Inclut le temps complet, mais tronque la partie fractionnaire des millisecondes, au format HH:MM:SS.sss.
- 'microseconds' : Inclut le temps complet, au format HH:MM:SS.mmmmmm.

Note : Les composants de temps exclus sont tronqués et non arrondis.

Une `ValueError` sera levée en cas d'argument *timespec* invalide.

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec=
↳ 'minutes')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

Nouveau dans la version 3.6 : Ajout de l'argument *timespec*.

`time.__str__()`

Pour un temps *t*, `str(t)` est équivalent à `t.isoformat()`.

`time.strftime(format)`

Renvoie une chaîne de caractères représentant la date, contrôlée par une chaîne de formatage explicite. Pour une liste complète des directives de formatage, voir *Comportement de strftime() et strptime()*.

`time.__format__(format)`

Identique à `time.strftime()`. Cela permet de spécifier une chaîne de formatage pour un objet *time* dans une chaîne de formatage littérale et à l'utilisation de `str.format()`. Pour une liste complète des directives de formatage, voir *Comportement de strftime() et strptime()*.

`time.utcoffset()`

Si *tzinfo* est `None`, renvoie `None`, sinon renvoie `self.tzinfo.utcoffset(None)`, et lève une exception si l'expression précédente ne renvoie pas `None` ou un objet *timedelta* représentant un nombre entier de minutes de magnitude inférieure à un jour.

`time.dst()`

Si *tzinfo* est `None`, renvoie `None`, sinon renvoie `self.tzinfo.dst(None)`, et lève une exception si l'expression précédente ne renvoie pas `None` ou un objet *timedelta* représentant un nombre entier de minutes de magnitude inférieure à un jour.

`time.tzname()`

Si *tzinfo* est `None`, renvoie `None`, sinon renvoie `self.tzinfo.tzname(None)`, et lève une exception si l'expression précédente ne renvoie pas `None` ou une chaîne de caractères.

Exemple :

```
>>> from datetime import time, tzinfo, timedelta
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self, dt):
...         return "Europe/Prague"
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t
```

(suite sur la page suivante)

(suite de la page précédente)

```

datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'

```

8.1.6 Objets `tzinfo`

`class datetime.tzinfo`

Cette classe est une classe abstraite, signifiant qu'elle ne doit pas être instanciée directement. Vous devez en dériver une sous-classe concrète, et (au minimum) fournir des implémentations aux méthodes standard `tzinfo` requises par les méthodes de `datetime` que vous utilisez. Le module `datetime` fournit une simple sous-classe concrète de `tzinfo`, `timezone`, qui peut représenter des fuseaux horaires avec des décalages fixes par rapport à UTC, tels qu'UTC lui-même ou les nord-américains EST et EDT.

Une instance (d'une sous-classe concrète) de `tzinfo` peut être passée aux constructeurs des objets `datetime` et `time`. Les objets en question voient leurs attributs comme étant en temps local, et l'objet `tzinfo` contient des méthodes pour obtenir le décalage du temps local par rapport à UTC, le nom du fuseau horaire, le décalage d'heure d'été, tous relatifs à un objet de date ou d'heure qui leur est passé.

Prérequis spécifique au *pickling* : Une sous-classe `tzinfo` doit avoir une méthode `__init__()` qui peut être appelée sans arguments, sans quoi un objet sérialisé ne pourrait pas toujours être désérialisé. C'est un prérequis technique qui pourrait être assoupli dans le futur.

Une sous-classe concrète de `tzinfo` peut devoir implémenter les méthodes suivantes. Les méthodes réellement nécessaires dépendent de l'utilisation qui est faite des objets `datetime` avisés. Dans le doute, implémentez-les toutes.

`tzinfo.utcoffset(dt)`

Renvoie le décalage entre le temps local et UTC, en minutes vers l'est d'UTC. Si le temps local se situe à l'ouest d'UTC, le décalage devrait être négatif. Notez que cela est prévu pour être le décalage total par rapport à UTC; par exemple, si un objet `tzinfo` représente à la fois un fuseau horaire et son ajustement à l'heure d'été, `utcoffset()` devrait renvoyer leur somme. Si le décalage UTC n'est pas connu, renvoie `None`. Sinon, la valeur renvoyée doit être un objet `timedelta` spécifiant un nombre entier de minutes dans l'intervalle de -1439 à 1439 inclus ($1440 = 24 \times 60$; la magnitude du décalage doit être inférieure à un jour). La plupart des implémentations de `utcoffset()` ressembleront probablement à l'une des deux suivantes :

```

return CONSTANT                # fixed-offset class
return CONSTANT + self.dst(dt) # daylight-aware class

```

Si `utcoffset()` ne renvoie pas `None`, `dst()` ne doit pas non plus renvoyer `None`.

L'implémentation par défaut de `utcoffset()` lève une `NotImplementedError`.

`tzinfo.dst(dt)`

Renvoie l'ajustement d'heure d'été (DST, *daylight saving time*), en minutes vers l'est d'UTC, ou `None` si l'information n'est pas connue. Renvoie `timedelta(0)` si l'heure d'été n'est pas effective. Si elle est effective, renvoie un décalage sous forme d'un objet `timedelta` (voir `utcoffset()` pour les détails). Notez que ce décalage, si applicable, est déjà compris dans le décalage UTC renvoyé par `utcoffset()`, il n'est donc pas nécessaire de faire appel à `dst()` à moins que vous ne souhaitiez obtenir les informations séparément. Par exemple, `datetime.timetuple()` appelle la méthode `dst()` de son attribut `tzinfo` pour déterminer si l'option `tm_isdst` doit

être activée, et `tzinfo.fromutc()` fait appel à `dst()` pour tenir compte des heures d'été quand elle traverse des fuseaux horaires.

Une instance `tz` d'une sous-classe `tzinfo` convenant à la fois pour une heure standard et une heure d'été doit être cohérente :

```
tz.utcoffset(dt) - tz.dst(dt)
```

doit renvoyer le même résultat pour tout objet `datetime dt` avec `dt.tzinfo == tz`. Pour les sous-classes saines de `tzinfo`, cette expression calcule le « décalage standard » du fuseau horaire, qui ne doit pas dépendre de la date ou de l'heure, mais seulement de la position géographique. L'implémentation de `datetime.astimezone()` se base là-dessus, mais ne peut pas détecter les violations ; il est de la responsabilité du programmeur de l'assurer. Si une sous-classe `tzinfo` ne le garantit pas, il doit être possible de redéfinir l'implémentation par défaut de `tzinfo.fromutc()` pour tout de même fonctionner correctement avec `astimezone()`.

La plupart des implémentations de `dst()` ressembleront probablement à l'une des deux suivantes :

```
def dst(self, dt):
    # a fixed-offset class: doesn't account for DST
    return timedelta(0)
```

ou :

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time. Then

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

L'implémentation par défaut de `dst()` lève une `NotImplementedError`.

`tzinfo.tzname(dt)`

Renvoie le nom du fuseau horaire correspondant à l'objet `datetime dt`, sous forme d'une chaîne de caractères. rien n'est défini sur les noms par le module `datetime`, et il n'est pas nécessaire que ces noms signifient quelque chose en particulier. Par exemple, « GMT », « UTC », « -500 », « -5:00 », « EDT », « US/Eastern » et « America/New York » sont toutes des valeurs de retour valides. Renvoie `None` si un nom est inconnu. Notez qu'il s'agit d'une méthode et non d'une chaîne fixée en amont, parce que les sous-classes de `tzinfo` peuvent souhaiter renvoyer des noms différents en fonction de valeurs de `dt` spécifiques, en particulier si la classe `tzinfo` tient compte de l'heure d'été.

L'implémentation par défaut de `tzname()` lève une `NotImplementedError`.

Ces méthodes sont appelées par les objets `datetime` et `time`, en réponse à leurs méthodes aux mêmes noms. Un objet `datetime` se passe lui-même en tant qu'argument, et un objet `time` passe `None`. Les méthodes des sous-classes `tzinfo` doivent alors être prêtes à recevoir un argument `None` pour `dt`, ou une instance de `datetime`.

Quand `None` est passé, il est de la responsabilité du designer de la classe de choisir la meilleure réponse. Par exemple, renvoyer `None` est approprié si la classe souhaite signaler que les objets de temps ne participent pas au protocole `tzinfo`. Il peut être plus utile pour `utcoffset(None)` de renvoyer le décalage UTC standard, comme il n'existe aucune autre convention pour obtenir ce décalage.

Quand un objet `datetime` est passé en réponse à une méthode de `datetime`, `dt.tzinfo` est le même objet que `self`. Les méthodes de `tzinfo` peuvent se baser là-dessus, à moins que le code utilisateur appelle directement des méthodes de `tzinfo`. L'intention est que les méthodes de `tzinfo` interprètent `dt` comme étant le temps local, et n'aient pas à se soucier des objets dans d'autres fuseaux horaires.

Il y a une dernière méthode de `tzinfo` que les sous-classes peuvent vouloir redéfinir :

`tzinfo.fromutc(dt)`

Elle est appelée par l'implémentation par défaut de `datetime.astimezone()`. Quand appelée depuis cette

méthode, `dt.tzinfo` est *self*, et les données de date et d'heure de *dt* sont vues comme exprimant un temps UTC. Le rôle de `fromutc()` est d'ajuster les données de date et d'heure, renvoyant un objet *datetime* équivalent à *self*, dans le temps local.

La plupart des sous-classes *tzinfo* doivent être en mesure d'hériter sans problème de l'implémentation par défaut de `fromutc()`. Elle est suffisamment robuste pour gérer les fuseaux horaires à décalage fixe, et les fuseaux représentant à la fois des heures standards et d'été, et ce même si le décalage de l'heure d'été est différent suivant les années. Un exemple de fuseau horaire qui ne serait pas géré correctement dans tous les cas par l'implémentation par défaut de `fromutc()` en est un où le décalage standard (par rapport à UTC) dépend de valeurs spécifiques de date et d'heure passées, ce qui peut arriver pour des raisons politiques. Les implémentations par défaut de `astimezone()` et `fromutc()` peuvent ne pas produire les résultats attendus si le résultat est l'une des heures affectées par le changement d'heure.

En omettant le code des cas d'erreurs, l'implémentation par défaut de `fromutc()` se comporte comme suit :

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst # this is self's standard offset
    if delta:
        dt += delta # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

Exemple de classes *tzinfo* :

```
from datetime import tzinfo, timedelta, datetime, timezone

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
# timezones where UTC offset and/or the DST rules had
# changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
```

(suite sur la page suivante)

(suite de la page précédente)

```

    # Detect fold
    fold = (args == _time.localtime(stamp - dst_diff))
    return datetime(*args, microsecond=dt.microsecond,
                    tzinfo=self, fold=fold)

def utcoffset(self, dt):
    if self._isdst(dt):
        return DSTOFFSET
    else:
        return STDOFFSET

def dst(self, dt):
    if self._isdst(dt):
        return DSTDIFF
    else:
        return ZERO

def tzname(self, dt):
    return _time.tzname[self._isdst(dt)]

def _isdst(self, dt):
    tt = (dt.year, dt.month, dt.day,
          dt.hour, dt.minute, dt.second,
          dt.weekday(), 0, 0)
    stamp = _time.mktime(tt)
    tt = _time.localtime(stamp)
    return tt.tm_isdst > 0

Local = LocalTimezone()

# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt

# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)

```

(suite sur la page suivante)

(suite de la page précédente)

```

DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end

class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them. The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)

```

(suite sur la page suivante)

(suite de la page précédente)

```

if start + HOUR <= dt < end - HOUR:
    # DST is in effect.
    return HOUR
if end - HOUR <= dt < end:
    # Fold (an ambiguous hour): use dt.fold to disambiguate.
    return ZERO if dt.fold else HOUR
if start <= dt < start + HOUR:
    # Gap (a non-existent hour): reverse the fold rule.
    return HOUR if dt.fold else ZERO
# DST is off.
return ZERO

def fromutc(self, dt):
    assert dt.tzinfo is self
    start, end = us_dst_range(dt.year)
    start = start.replace(tzinfo=self)
    end = end.replace(tzinfo=self)
    std_time = dt + self.stdoffset
    dst_time = std_time + HOUR
    if end <= dst_time < end + HOUR:
        # Repeated hour
        return std_time.replace(fold=1)
    if std_time < start or dst_time >= end:
        # Standard time
        return std_time
    if start <= std_time < end - HOUR:
        # Daylight saving time
        return dst_time

```

```

Eastern = USTimeZone(-5, "Eastern", "EST", "EDT")
Central = USTimeZone(-6, "Central", "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific = USTimeZone(-8, "Pacific", "PST", "PDT")

```

Notez que, deux fois par an, on rencontre des subtilités inévitables dans les sous-classes de `tzinfo` représentant à la fois des heures standard et d'été, au passage de l'une à l'autre. Concrètement, considérez le fuseau de l'est des États-Unis (UTC -0500), où EDT (heure d'été) débute à la minute qui suit 1 :59 (EST) le second dimanche de mars, et se termine à la minute qui suit 1 :59 (EDT) le premier dimanche de novembre :

```

UTC      3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
EST     22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
EDT     23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start   22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

end     23:MM  0:MM  1:MM  1:MM  2:MM  3:MM

```

Quand l'heure d'été débute (la ligne « *start* »), l'horloge locale passe de 1 :59 à 3 :00. Une heure de la forme 2 :MM n'a pas vraiment de sens ce jour là, donc `astimezone` (Eastern) ne délivrera pas de résultat avec `hour == 2` pour le jour où débute l'heure d'été. Par exemple, lors de la transition du printemps 2016, nous obtenons

```

>>> u0 = datetime(2016, 3, 13, 5, tzinfo=tztimezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)

```

(suite sur la page suivante)

(suite de la page précédente)

```
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

Quand l'heure d'été se termine (la ligne « *end* »), il y a potentiellement un problème pire que cela : il y a une heure qui ne peut pas être exprimée sans ambiguïté en temps local : la dernière heure de l'heure d'été. Dans l'est des États-Unis, l'heure d'été se termine sur les heures de la forme 5 :MM UTC. L'horloge locale passe de 1 :59 (heure d'été) à 1 :00 (heure standard) à nouveau. Les heures locales de la forme 1 :MM sont ambiguës. `astimezone()` imite le comportement des horloges locales en associant deux heures UTC adjacentes à la même heure locale. Dans notre exemple, les temps UTC de la forme 5 :MM et 6 :MM sont tous deux associés à 1 :MM quand convertis vers ce fuseau, mais les heures les plus anciennes ont l'attribut `fold` à 0 et les plus récentes l'ont à 1. Par exemple, lors de la transition de l'automne 2016, nous obtenons

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

Notez que deux instances `datetime` qui ne diffèrent que par la valeur de leur attribut `fold` sont considérées égales dans les comparaisons.

Les applications qui ne peuvent pas gérer ces ambiguïtés doivent vérifier explicitement la valeur de l'attribut `fold` ou éviter d'utiliser des sous-classes `tzinfo` hybrides ; il n'y a aucune ambiguïté lors de l'utilisation de la classe `timezone`, ou toute autre sous-classe de `tzinfo` à décalage fixe (comme une classe représentant uniquement le fuseau EST (de décalage fixe -5h) ou uniquement EDT (-4h)).

Voir aussi :

dateutil.tz La bibliothèque standard contient la classe `timezone` pour gérer des décalages fixes par rapport à UTC et `timezone.utc` comme instance du fuseau horaire UTC.

La bibliothèque `dateutils.tz` apporte à Python la *base de données de fuseaux horaires IANA* (*IANA timezone database*, aussi appelée base de données Olson) , et son utilisation est recommandée.

Base de données des fuseaux horaires de l'IANA La *Time Zone Database* (souvent appelée *tz*, *tzdata* ou *zoneinfo*) contient les codes et les données représentant l'historique du temps local pour un grand nombre d'emplacements représentatifs autour du globe. Elle est mise à jour périodiquement, pour refléter les changements opérés par des politiques sur les bornes du fuseau, les décalages UTC, et les règles de passage à l'heure d'été.

8.1.7 Objets `timezone`

La classe `timezone` est une sous-classe de `tzinfo`, où chaque instance représente un fuseau horaire défini par un décalage fixe par rapport à UTC. Notez que les objets de cette classe ne peuvent pas être utilisés pour représenter les informations de fuseaux horaires dans des emplacements où plusieurs décalages sont utilisés au cours de l'année ou où des changements historiques ont été opérés sur le temps civil.

class `datetime.timezone` (*offset*, *name=None*)

L'argument *offset* doit être spécifié comme un objet `timedelta` représentant la différence entre le temps local et UTC. Il doit être strictement compris entre `-timedelta(hours=24)` et `timedelta(hours=24)` et représenter un nombre entier de minutes, autrement une `ValueError` est levée.

L'argument *name* est optionnel. Si spécifié, il doit être une chaîne de caractères qui sera utilisée comme valeur de retour de la méthode `datetime.tzname()`.

Nouveau dans la version 3.2.

`timezone.utcoffset` (*dt*)

Renvoie la valeur fixe spécifiée à la création de l'instance de `timezone`. L'argument *dt* est ignoré. La valeur de retour est une instance `timedelta` égale à la différence entre le temps local et UTC.

`timezone.tzname` (*dt*)

Renvoie la valeur fixe spécifiée à la création de l'instance de `timezone`. Si *name* n'est pas fourni au constructeur, le nom renvoyé par `tzname(dt)` est généré comme suit à partir de la valeur de *offset*. Si *offset* vaut `timedelta(0)`, le nom sera « UTC », autrement le nom sera une chaîne de la forme « UTC±HH:MM », où ± est le signe d'offset, et HH et MM sont respectivement les représentations à deux chiffres de `offset.hours` et `offset.minutes`.

Modifié dans la version 3.6 : Le nom généré à partir de `offset=timedelta(0)` est maintenant « UTC » plutôt que « UTC+00:00 ».

`timezone.dst` (*dt*)

Renvoie toujours `None`.

`timezone.fromutc` (*dt*)

Renvoie `dt + offset`. L'argument *dt* doit être une instance avisée de `datetime`, avec `tzinfo` valant `self`.

Attributs de la classe :

`timezone.utc`

Le fuseau horaire UTC, `timezone(timedelta(0))`.

8.1.8 Comportement de `strftime()` et `strptime()`

Les objets `date`, `datetime` et `time` comportent tous une méthode `strftime(format)`, pour créer une représentation du temps sous forme d'une chaîne de caractères, contrôlée par une chaîne de formatage explicite. Grossièrement, `d.strftime(fmt)` se comporte comme la fonction `time.strftime(fmt, d.timetuple())` du module `time`, bien que tous les objets ne comportent pas de méthode `timetuple()`.

Conversely, the `datetime.strptime()` class method creates a `datetime` object from a string representing a date and time and a corresponding format string. `datetime.strptime(date_string, format)` is equivalent to `datetime(*(time.strptime(date_string, format)[0:6]))`, except when the format includes sub-second components or timezone offset information, which are supported in `datetime.strptime` but are discarded by `time.strptime`.

Pour les objets `time`, les codes de formatage pour l'année, le mois et le jour ne devraient pas être utilisés, puisque les objets de temps ne possèdent pas de telles valeurs. S'ils sont tout de même utilisés, 1900 est substitué à l'année, et 1 au mois et au jour.

Pour les objets `date`, les codes de formatage pour les heures, minutes, secondes et microsecondes ne devraient pas être utilisés, puisque les objets `date` ne possèdent pas de telles valeurs. S'ils sont tous de même utilisés, ils sont substitués par 0.

L'ensemble complet des codes de formatage implémentés varie selon les plateformes, parce que Python appelle la fonction `strftime()` de la bibliothèque C de la plateforme, et les variations sont courantes. Pour voir un ensemble complet des codes de formatage implémentés par votre plateforme, consultez la documentation de `strftime(3)`.

La liste suivante est la liste de tous les codes de formatage requis par le standard C (version 1989), ils fonctionnent sur toutes les plateformes possédant une implémentation de C standard. Notez que la version 1999 du standard C a ajouté des codes de formatage additionnels.

Directive	Signification	Exemple	Notes
%a	Jour de la semaine abrégé dans la langue locale.	Sun, Mon, ..., Sat (en_US); Lu, Ma, ..., Di (fr_FR)	(1)
%A	Jour de la semaine complet dans la langue locale.	Sunday, Monday, ..., Saturday (en_US); Lundi, Mardi, ..., Dimanche (fr_FR)	(1)
%w	Jour de la semaine en chiffre, avec 0 pour le dimanche et 6 pour le samedi.	0, 1, ..., 6	
%d	Jour du mois sur deux chiffres.	01, 02, ..., 31	
%b	Nom du mois abrégé dans la langue locale.	Jan, Feb, ..., Dec (en_US); janv., févr., ..., déc. (fr_FR)	(1)
%B	Nom complet du mois dans la langue locale.	January, February, ..., December (en_US); janvier, février, ..., décembre (fr_FR)	(1)
%m	Numéro du mois sur deux chiffres.	01, 02, ..., 12	
%y	Année sur deux chiffres (sans le siècle).	00, 01, ..., 99	
%Y	Année complète sur quatre chiffres.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
%H	Heure à deux chiffres de 00 à 23.	00, 01, ..., 23	
%I	Heure à deux chiffres pour les horloges 12h (01 à 12).	01, 02, ..., 12	
%p	Équivalent local à AM/PM.	AM, PM (en_US); am, pm (de_DE)	(1), (3)
%M	Minutes sur deux chiffres.	00, 01, ..., 59	
%S	Secondes sur deux chiffres.	00, 01, ..., 59	(4)
%f	Microsecondes sur 6 chiffres.	000000, 000001, ..., 999999	(5)
%z	Décalage UTC sous la forme +HHMM ou -HHMM (chaîne vide si non défini).	(vide), +0000, -0400, +1030	(6)
8.1. datetime — Types de base pour la date et l'heure			197
%Z	Nom du fuseau horaire (chaîne vide si l'instance est naïve).	(vide), UTC, EST, CST	

Plusieurs directives additionnelles non requises par le standard C89 sont incluses par commodité. Ces paramètres correspondent tous aux valeurs de dates ISO 8601. Ils peuvent ne pas être disponibles sur toutes les plateformes quand utilisés avec la méthode `strptime()`. Les directives ISO 8601 d'année et de semaine ne sont pas interchangeables avec les directives d'année et de semaine précédentes. Appeler `strptime()` avec des directives ISO 8601 incomplètes ou ambiguës lèvera une `ValueError`.

Directive	Signification	Exemple	Notes
%G	Année complète ISO 8601 représentant l'année contenant la plus grande partie de la semaine ISO (%V).	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(8)
%u	Jour de la semaine ISO 8601 où 1 correspond au lundi.	1, 2, ..., 7	
%V	Numéro de la semaine ISO 8601, avec lundi étant le premier jour de la semaine. La semaine 01 est la semaine contenant le 4 janvier.	01, 02, ..., 53	(8)

Nouveau dans la version 3.6 : %G, %u et %V ont été ajoutés.

Notes :

- (1) Comme le format dépend de la locale courante, les assumptions sur la valeur de retour doivent être prises soigneusement. L'ordre des champs variera (par exemple, « mois/jour/année » versus « année/mois/jour »), et le retour pourrait contenir des caractères Unicode encodés en utilisant l'encodage par défaut de la locale (par exemple, si la locale courante est `ja_JP`, l'encodage par défaut pourrait être `eucJP`, `SJIS` ou `utf-8` ; utilisez `locale.getlocale()` pour déterminer l'encodage de la locale courante).
- (2) La méthode `strptime()` peut analyser toutes les années de l'intervalle [1, 9999], mais toutes les années < 1000 doivent être représentées sur quatre chiffres.
Modifié dans la version 3.2 : Dans les versions précédentes, la méthode `strptime()` était limitée aux années >= 1900.
Modifié dans la version 3.3 : En version 3.2, la méthode `strptime()` était limitée aux années >= 1000.
- (3) Quand utilisée avec la méthode `strptime()`, la directive %p n'affecte l'heure extraite que si la directive %I est utilisée pour analyser l'heure.
- (4) À l'inverse du module `time`, le module `datetime` ne gère pas les secondes intercalaires.
- (5) Quand utilisée avec la méthode `strptime()`, la directive %f accepte un nombre de 1 à 6 chiffres, où des zéros seront ajoutés à droite jusqu'à former un nombre de 6 chiffres. %f est une extension de l'ensemble des caractères de formatage du standard C (mais implémentée séparément dans les objets `datetime`, la rendant ainsi toujours disponible).
- (6) Pour les objets naïfs, les codes de formatage %z et %Z sont remplacés par des chaînes vides.
Pour un objet avisé :
%z Le résultat de `utcoffset()` est transformé en une chaîne de 5 caractères de la forme +HHMM ou -HHMM, où HH est une chaîne de deux chiffres donnant le nombre d'heures du décalage UTC, et MM est une chaîne de deux chiffres donnant le nombre de minutes de ce décalage. Par exemple, si `utcoffset()` renvoie `timedelta(hours=-3, minutes=-30)`, %z est remplacé par la chaîne "-0330".
%Z Si `tzname()` renvoie `None`, %Z est remplacé par une chaîne vide. Autrement %Z est remplacé par la valeur renvoyée, qui doit être une chaîne.
Modifié dans la version 3.2 : Quand la directive %z est fournie à la méthode `strptime()`, un objet `datetime` avisé est construit. L'attribut `tzinfo` du résultat aura pour valeur une instance de `timezone`.
- (7) Quand ces directives sont utilisées avec la méthode `strptime()`, %U et %W ne sont utilisées dans les calculs que si le jour de la semaine et l'année calendaire (%Y) sont spécifiés.
- (8) De façon similaire à %U et %W, %v n'est utilisé dans les calculs que lorsque le jour de la semaine et l'année ISO (%G) sont spécifiés dans la chaîne de formatage `strptime()`. Notez aussi que %G et %Y ne sont pas interchangeables.

Notes

8.2 calendar — General calendar-related functions

Code source : [Lib/calendar.py](#)

Ce module permet d'afficher un calendrier comme le fait le programme Unix `cal`, et il fournit des fonctions utiles relatives au calendrier. Par défaut, ces calendriers ont le lundi comme premier jour de la semaine et le dimanche comme dernier jour. Utilisez `setfirstweekday()` pour définir le premier jour de la semaine à dimanche (6) ou à tout autre jour de la semaine. Les paramètres pour spécifier les dates sont donnés sous forme de nombres entiers. Voir aussi les modules `datetime` et `time`.

Most of these functions and classes rely on the `datetime` module which uses an idealized calendar, the current Gregorian calendar extended in both directions. This matches the definition of the « proleptic Gregorian » calendar in Dershowitz and Reingold's book « Calendrical Calculations », where it's the base calendar for all computations.

class `calendar.Calendar` (*firstweekday=0*)

Crée un objet `Calendar`. *firstweekday* est un entier spécifiant le premier jour de la semaine, valant par défaut 0 (lundi), pouvant aller jusqu'à 6 (dimanche).

L'objet `Calendar` fournit plusieurs méthodes pouvant être utilisées pour préparer les données du calendrier pour le formatage. Cette classe ne fait pas de formatage elle-même. Il s'agit du travail des sous-classes.

Les instances de `Calendar` ont les méthodes suivantes :

iterweekdays ()

Renvoie un itérateur sur les numéros des jours d'une semaine. La première valeur est donc la même que la valeur de la propriété `firstweekday`.

itermonthdates (*year, month*)

Renvoie un itérateur sur les jours du mois *month* (1 à 12) de l'année *year*. Cet itérateur renvoie tous les jours du mois (sous forme d'instances de `datetime.date`) ainsi que tous les jours avant le début du mois et après la fin du mois nécessaires pour obtenir des semaines complètes.

itermonthdays2 (*year, month*)

Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`. Days returned will be tuples consisting of a day number and a week day number.

itermonthdays (*year, month*)

Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`. Days returned will simply be day numbers.

monthdatescalendar (*year, month*)

Renvoie la liste des semaines complètes du mois *month* de l'année *year*. Les semaines sont des listes de sept objets `datetime.date`.

monthdays2calendar (*year, month*)

Renvoie la liste des semaines complètes du mois *month* de l'année *year*. Les semaines sont des listes de sept paires contenant le numéro du jour dans le mois et du numéro du jour dans la semaine.

monthdayscalendar (*year, month*)

Renvoie la liste des semaines complètes du mois *month* de l'année *year*. Les semaines sont une liste de sept numéros de jours.

yeardatescalendar (*year, width=3*)

Renvoie ce qu'il faut pour afficher correctement une année. La valeur renvoyée est une liste de lignes de mois. Chaque ligne mensuelle contient jusqu'à *width* mois (avec une valeur par défaut à 3). Chaque mois contient de 4 à 6 semaines et chaque semaine 1 à 7 jours. Les jours sont des objets `datetime.date`.

yeardays2calendar (*year, width=3*)

Renvoie ce qu'il faut pour afficher correctement une année, (similaire à `yeardatescalendar()`). Les listes des semaines contiennent des paires contenant le numéro du jour du mois et le numéro du jour de la semaine. Les numéros des jours en dehors de ce mois sont à zéro.

yeardayscalendar (*year*, *width*=3)

Renvoie ce qu'il faut pour afficher correctement une année, (similaire à `yeardatescalendar()`). Les listes de semaines contiennent des numéros de jours. Les numéros de jours en dehors de ce mois sont de zéro.

class `calendar.TextCalendar` (*firstweekday*=0)

Cette classe peut être utilisée pour générer des calendriers en texte brut.

Les instances `TextCalendar` exposent les méthodes suivantes :

formatmonth (*theyear*, *themonth*, *w*=0, *l*=0)

Donne le calendrier d'un mois dans une chaîne multi-ligne. Si *w* est fourni, il spécifie la largeur des colonnes de date, qui sont centrées. Si *l* est donné, il spécifie le nombre de lignes que chaque semaine utilisera. Le résultat varie en fonction du premier jour de la semaine spécifié dans le constructeur ou défini par la méthode `setfirstweekday()`.

prmonth (*theyear*, *themonth*, *w*=0, *l*=0)

Affiche le calendrier d'un mois tel que renvoyé par `formatmonth()`.

formatyear (*theyear*, *w*=2, *l*=1, *c*=6, *m*=3)

Renvoie un calendrier de *m* colonnes pour une année entière sous forme de chaîne multi-ligne. Les paramètres facultatifs *w*, *l* et *c* correspondent respectivement à la largeur de la colonne date, les lignes par semaines, le nombre d'espace entre les colonnes de mois. Le résultat varie en fonction du premier jour de la semaine spécifié dans le constructeur ou défini par la méthode `setfirstweekday()`. La première année pour laquelle un calendrier peut être généré, dépend de la plateforme.

pryear (*theyear*, *w*=2, *l*=1, *c*=6, *m*=3)

Affiche le calendrier pour une année entière comme renvoyé par `formatyear()`.

class `calendar.HTMLCalendar` (*firstweekday*=0)

Cette classe peut être utilisée pour générer des calendriers HTML.

`HTMLCalendar` instances have the following methods :

formatmonth (*theyear*, *themonth*, *withyear*=True)

Return a month's calendar as an HTML table. If *withyear* is true the year will be included in the header, otherwise just the month name will be used.

formatyear (*theyear*, *width*=3)

Return a year's calendar as an HTML table. *width* (defaulting to 3) specifies the number of months per row.

formatyearpage (*theyear*, *width*=3, *css*='calendar.css', *encoding*=None)

Return a year's calendar as a complete HTML page. *width* (defaulting to 3) specifies the number of months per row. *css* is the name for the cascading style sheet to be used. `None` can be passed if no style sheet should be used. *encoding* specifies the encoding to be used for the output (defaulting to the system default encoding).

class `calendar.LocaleTextCalendar` (*firstweekday*=0, *locale*=None)

This subclass of `TextCalendar` can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

class `calendar.LocaleHTMLCalendar` (*firstweekday*=0, *locale*=None)

This subclass of `HTMLCalendar` can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

Note : The `formatweekday()` and `formatmonthname()` methods of these two classes temporarily change the current locale to the given *locale*. Because the current locale is a process-wide setting, they are not thread-safe.

For simple text calendars this module provides the following functions.

`calendar.setfirstweekday` (*weekday*)

Sets the weekday (0 is Monday, 6 is Sunday) to start each week. The values MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, and SUNDAY are provided for convenience. For example, to set the first weekday to Sunday :

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

`calendar.firstweekday()`

Returns the current setting for the weekday to start each week.

`calendar.isleap(year)`

Returns *True* if *year* is a leap year, otherwise *False*.

`calendar.leapdays(y1, y2)`

Returns the number of leap years in the range from *y1* to *y2* (exclusive), where *y1* and *y2* are years.

This function works for ranges spanning a century change.

`calendar.weekday(year, month, day)`

Returns the day of the week (0 is Monday) for *year* (1970–...), *month* (1–12), *day* (1–31).

`calendar.weekheader(n)`

Return a header containing abbreviated weekday names. *n* specifies the width in characters for one weekday.

`calendar.monthrange(year, month)`

Returns weekday of first day of the month and number of days in month, for the specified *year* and *month*.

`calendar.monthcalendar(year, month)`

Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month are represented by zeros. Each week begins with Monday unless set by `setfirstweekday()`.

`calendar.prmonth(theyear, themonth, w=0, l=0)`

Prints a month's calendar as returned by `month()`.

`calendar.month(theyear, themonth, w=0, l=0)`

Returns a month's calendar in a multi-line string using the `formatmonth()` of the `TextCalendar` class.

`calendar.prcal(year, w=0, l=0, c=6, m=3)`

Prints the calendar for an entire year as returned by `calendar()`.

`calendar.calendar(year, w=2, l=1, c=6, m=3)`

Returns a 3-column calendar for an entire year as a multi-line string using the `formatyear()` of the `TextCalendar` class.

`calendar.timegm(tuple)`

An unrelated but handy function that takes a time tuple such as returned by the `gmtime()` function in the `time` module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, `time.gmtime()` and `timegm()` are each others' inverse.

The `calendar` module exports the following data attributes :

`calendar.day_name`

An array that represents the days of the week in the current locale.

`calendar.day_abbr`

An array that represents the abbreviated days of the week in the current locale.

`calendar.month_name`

An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_name[0]` is the empty string.

`calendar.month_abbr`

An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_abbr[0]` is the empty string.

Voir aussi :

Module `datetime` Object-oriented interface to dates and times with similar functionality to the `time` module.

Module `time` Low-level time related functions.

8.3 collections — Types de données de conteneurs

Code source : `Lib/collections/__init__.py`

Ce module implémente des types de données de conteneurs spécialisés qui apportent des alternatives aux conteneurs natifs de Python plus généraux `dict`, `list`, `set` et `tuple`.

<code>namedtuple()</code>	fonction permettant de créer des sous-classes de <code>tuple</code> avec des champs nommés
<code>deque</code>	conteneur se comportant comme une liste avec des ajouts et retraits rapides à chaque extrémité
<code>ChainMap</code>	classe semblable aux dictionnaires qui crée une unique vue à partir de plusieurs dictionnaires
<code>Counter</code>	sous-classe de <code>dict</code> pour compter des objets hachables
<code>OrderedDict</code>	sous-classe de <code>dict</code> qui garde en mémoire l'ordre dans lequel les entrées ont été ajoutées
<code>defaultdict</code>	sous-classe de <code>dict</code> qui appelle une fonction de fabrication en cas de valeur manquante
<code>UserDict</code>	surcouche autour des objets dictionnaires pour faciliter l'héritage de <code>dict</code>
<code>UserList</code>	surcouche autour des objets listes pour faciliter l'héritage de <code>list</code>
<code>UserString</code>	surcouche autour des objets chaînes de caractères pour faciliter l'héritage de <code>str</code>

Modifié dans la version 3.3 : Moved *Classes de base abstraites de collections* to the `collections.abc` module. For backwards compatibility, they continue to be visible in this module as well.

8.3.1 Objets ChainMap

Nouveau dans la version 3.3.

Le module fournit une classe `ChainMap` afin de réunir rapidement plusieurs dictionnaires en une unique entité. Cela est souvent plus rapide que de créer un nouveau dictionnaire et d'effectuer plusieurs appels de `update()`.

Cette classe peut être utilisée pour simuler des portées imbriquées, elle est aussi utile pour le *templating*.

class `collections.ChainMap` (**maps*)

Un objet `ChainMap` regroupe plusieurs dictionnaires (ou autres tableaux de correspondance) en une vue que l'on peut mettre à jour. Si le paramètre *maps* est vide, un dictionnaire vide est fourni de telle manière qu'une nouvelle chaîne possède toujours au moins un dictionnaire.

Les dictionnaires sous-jacents sont stockés dans une liste. Celle-ci est publique et peut être consultée ou mise à jour via l'attribut *maps*. Il n'y a pas d'autre état.

Les recherches s'effectuent successivement dans chaque dictionnaire jusqu'à la première clé correspondante. En revanche, les écritures, mises à jour et suppressions n'affectent que le premier dictionnaire.

Un objet `ChainMap` incorpore les dictionnaires sous-jacents par leur référence. Ainsi, si l'un d'eux est modifié, les changements affectent également la `ChainMap`.

Toutes les méthodes usuelles des dictionnaires sont gérées. De plus, cette classe fournit un attribut *maps*, une méthode pour créer de nouveaux sous-contextes et une propriété pour accéder à tous les dictionnaires sous-jacents excepté le premier :

maps

Liste de dictionnaires éditable par l'utilisateur et classée selon l'ordre de recherche. Il s'agit de l'unique état stocké et elle peut être modifiée pour changer l'ordre de recherche. La liste doit toujours contenir au moins un dictionnaire.

new_child (*m=None*)

Renvoie un nouvel objet *ChainMap* contenant un nouveau dictionnaire suivi par tous les autres de l'instance actuelle. Si *m* est spécifié, il devient le nouveau dictionnaire au début de la liste ; sinon, un dictionnaire vide est utilisé, de telle manière qu'appeler `d.new_child()` équivaut à appeler `ChainMap({}, *d.maps)`. Cette méthode est utile pour créer des sous-contextes qui peuvent être mis à jour sans altérer les valeurs dans les dictionnaires parents.

Modifié dans la version 3.4 : Ajout du paramètre optionnel *m*.

parents

Propriété qui renvoie un nouvel objet *ChainMap* contenant tous les dictionnaires de l'instance actuelle hormis le premier. Cette propriété est utile pour ignorer le premier dictionnaire dans les recherches ; son utilisation rappelle le mot-clé `nonlocal` (utilisé pour les *portées imbriquées*), ou bien la fonction native `super()`. Une référence à `d.parents` est équivalente à `:ChainMap(*d.maps[1:])`.

Voir aussi :

- La classe *MultiContext* dans le package *CodeTools* d'Enthought possède des options pour gérer l'écriture dans n'importe quel dictionnaire de la chaîne.
- Django's *Context* class for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the `new_child()` method and the `parents()` property.
- Le Cas pratique des contextes imbriqués a des options pour contrôler si les écritures et autres mutations ne s'appliquent qu'au premier ou à un autre dictionnaire de la chaîne.
- Une version grandement simplifiée de *Chainmap* en lecture seule.

Exemples et cas pratiques utilisant ChainMap

Cette partie montre diverses approches afin de travailler avec les dictionnaires chaînés.

Exemple 1 : simulation de la chaîne de recherche interne de Python :

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Exemple 2 : spécification d'une hiérarchie pour les options : ligne de commande, variable d'environnement, valeurs par défaut :

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k:v for k, v in vars(namespace).items() if v}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Exemple 3 : modèles pour simuler des contexte imbriqués avec la classe *ChainMap* :

```
c = ChainMap()           # Create root context
d = c.new_child()        # Create nested child context
e = c.new_child()        # Child of c, independent from d
e.maps[0]                # Current context dictionary -- like Python's locals()
e.maps[-1]               # Root context -- like Python's globals()
```

(suite sur la page suivante)

(suite de la page précédente)

```

e.parents          # Enclosing context chain -- like Python's nonlocals

d['x']              # Get first key in the chain of contexts
d['x'] = 1          # Set value in current context
del d['x']          # Delete from current context
list(d)            # All nested values
k in d             # Check all nested values
len(d)             # Number of nested values
d.items()          # All nested items
dict(d)            # Flatten into a regular dictionary

```

La classe `ChainMap` ne met à jour (écriture et suppression) que le premier dictionnaire de la chaîne, alors qu'une recherche inspecte toute la chaîne. Cependant, si l'on veut effectuer des écritures ou suppressions en profondeur, on peut facilement faire une sous-classe qui met à jour les clés trouvées de la chaîne en profondeur :

```

class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'          # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                           # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})

```

8.3.2 Objets Counter

Ce module fournit un outil pour effectuer rapidement et facilement des dénombrements. Par exemple :

```

>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)

```

(suite sur la page suivante)

(suite de la page précédente)

```
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554), ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class `collections.Counter` (*[iterable-or-mapping]*)

La classe `Counter` est une sous-classe de `dict` qui permet le dénombrement d'objets hachables. Il s'agit d'une collection non ordonnée dans laquelle les éléments sont stockés comme des clés de dictionnaire et leurs nombres d'occurrences respectifs comme leurs valeurs. Ceux-ci peuvent être des entiers relatifs (positifs, négatifs ou nuls). La classe `Counter` est similaire aux sacs ou aux multiensembles dans d'autres langages.

Les éléments sont comptés à partir d'un itérable ou initialisés à partir d'un autre dictionnaire (ou compteur) :

```
>>> c = Counter()                                # a new, empty counter
>>> c = Counter('gallahad')                      # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})           # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)                 # a new counter from keyword args
```

Les objets `Counter` ont une interface de dictionnaire, à l'exception près qu'ils renvoient zéro au lieu de lever une exception `KeyError` pour des éléments manquants :

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                                    # count of a missing element is zero
0
```

Mettre un comptage à zéro pour un élément ne le retire pas de l'objet `Counter`. Il faut utiliser `del` pour le supprimer complètement :

```
>>> c['sausage'] = 0                             # counter entry with a zero count
>>> del c['sausage']                             # del actually removes the entry
```

Nouveau dans la version 3.1.

En plus des méthodes disponibles pour tous les dictionnaires, les objets compteurs gèrent trois méthodes supplémentaires :

elements ()

Renvoie un itérateur sur chaque élément en le répétant autant de fois que la valeur du compteur associé. Les éléments sont renvoyés dans un ordre arbitraire. Si le comptage d'un élément est strictement inférieur à 1, alors `elements()` l'ignore.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common ([*n*])

Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or None, `most_common()` returns *all* elements in the counter. Elements with equal counts are ordered arbitrarily :

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

subtract (*[iterable-or-mapping]*)

Les éléments sont soustraits à partir d'un itérable ou d'un autre dictionnaire (ou compteur). Cette méthode se comporte comme `dict.update()` mais soustrait les nombres d'occurrences au lieu de les remplacer. Les entrées et sorties peuvent être négatives ou nulles.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Nouveau dans la version 3.2.

Les méthodes usuelles des dictionnaires sont disponibles pour les objets *Counter* à l'exception de deux méthodes qui fonctionnent différemment pour les compteurs.

fromkeys (*iterable*)

Cette méthode de classe n'est pas implémentée pour les objets *Counter*.

update ([*iterable-or-mapping*])

Les éléments sont comptés à partir d'un itérable ou ajoutés d'un autre dictionnaire (ou compteur). Cette méthode se comporte comme *dict.update()* mais additionne les nombres d'occurrences au lieu de les remplacer. De plus, l'itérable doit être une séquence d'éléments et non une séquence de paires (clé, valeur).

Opérations usuelles sur les objets *Counter* :

```
sum(c.values())           # total of all counts
c.clear()                 # reset all counts
list(c)                   # list unique elements
set(c)                    # convert to a set
dict(c)                   # convert to a regular dictionary
c.items()                 # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs)) # convert from a list of (elem, cnt) pairs
c.most_common()[:n-1:-1]  # n least common elements
+c                         # remove zero and negative counts
```

Quelques opérations mathématiques sont fournies pour combiner des objets *Counter* afin de créer des multiensembles (des compteurs dont les dénombrements des éléments sont strictement supérieurs à zéro). Les additions et soustractions combinent les compteurs en ajoutant ou retranchant les nombres d'occurrences des éléments correspondants. Les intersections et unions renvoient les minimums et maximums des comptages correspondants. Chaque opération peut accepter des entrées avec des comptages relatifs, mais la sortie exclut les résultats avec des comptages négatifs ou nuls.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                      # add two counters together: c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                      # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                      # intersection: min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                      # union: max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

L'addition et la soustraction unaires (avec un seul terme) sont des raccourcis pour respectivement additionner un compteur avec un compteur vide ou et pour retrancher un compteur d'un compteur vide.

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

Nouveau dans la version 3.3 : Ajout de la gestion des additions et soustractions unaires, et des remplacements dans les multiensembles.

Note : Les compteurs ont été conçus essentiellement pour fonctionner avec des entiers naturels pour représenter les

dénombrements en cours ; cependant, les cas d'utilisation nécessitant d'autres types ou des valeurs négatives n'ont pas été écartés. Pour vous aider dans ces cas particuliers, cette section documente la plage minimale et les restrictions de type.

- La classe `Counter` est elle-même une sous-classe de dictionnaire sans restriction particulière sur ces clés ou valeurs. Les valeurs ont vocation à être des nombres représentant des comptages, mais il est *possible* de stocker n'importe quel type de valeur.
- The `most_common()` method requires only that the values be orderable.
- For in-place operations such as `c[key] += 1`, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for `update()` and `subtract()` which allow negative and zero values for both inputs and outputs.
- Les méthodes de multiensembles sont uniquement conçues pour les cas d'utilisation avec des valeurs positives. Les entrées peuvent contenir des valeurs négatives ou nulles, mais seules les sorties avec des valeurs positives sont créées. Il n'y a pas de restriction de type, mais les types des valeurs doivent gérer l'addition, la soustraction et la comparaison.
- The `elements()` method requires integer counts. It ignores zero and negative counts.

Voir aussi :

- `Bag class` dans Smalltalk.
- L'article Wikipédia sur les [multiensembles](#) sur Wikipédia (ou l'article en anglais).
- Des guides et exemples à propos des [multiensembles en C++](#).
- Pour les opérations mathématiques sur les multiensembles et leurs applications, voir *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19.*
- To enumerate all distinct multisets of a given size over a given set of elements, see `itertools.combinations_with_replacement()` :

```
map(Counter, combinations_with_replacement("ABC", 2)) -> AA AB AC BB BC CC
```

8.3.3 Objets deque

class `collections.deque` (`[iterable[, maxlen]]`)

Renvoie un nouvel objet *deque* initialisé de gauche à droite (en utilisant `append()`) avec les données d'*iterable*. Si *iterable* n'est pas spécifié, alors la nouvelle *deque* est vide.

Les *deques* sont une généralisation des piles et des files (*deque* se prononce « *dèque* » et est l'abréviation de l'anglais *double-ended queue*) : il est possible d'ajouter et retirer des éléments par les deux bouts des *deques*. Celles-ci gèrent des ajouts et des retraits utilisables par de multiples fils d'exécution (*thread-safe*) et efficaces du point de vue de la mémoire des deux côtés de la *deque*, avec approximativement la même performance en $O(1)$ dans les deux sens.

Bien que les objets `list` gèrent des opérations similaires, ils sont optimisés pour des opérations qui ne changent pas la taille de la liste. Les opérations `pop(0)` et `insert(0, v)` qui changent la taille et la position de la représentation des données sous-jacentes entraînent des coûts de déplacement de mémoire en $O(n)$.

Si *maxlen* n'est pas spécifié ou vaut *None*, les *deques* peuvent atteindre une taille arbitraire. Sinon, la *deque* est limitée par cette taille maximale. Une fois que celle-ci est atteinte, un ajout d'un ou plusieurs éléments engendre la suppression du nombre correspondant d'éléments à l'autre extrémité de la *deque*. Les *deques* à longueur limitée apportent des fonctionnalités similaire au filtre `tail` d'Unix. Elles sont aussi utiles pour le suivi de transactions et autres lots de données où seule l'activité récente est intéressante.

Les objets *deques* gèrent les méthodes suivantes :

append (*x*)

Ajoute *x* à l'extrémité droite de la *deque*.

appendleft (*x*)

Ajoute *x* à l'extrémité gauche de la *deque*.

clear ()

Supprime tous les éléments de la *deque* et la laisse avec une longueur de 0.

copy()

Crée une copie superficielle de la *deque*.

Nouveau dans la version 3.5.

count(x)

Compte le nombre d'éléments de la *deque* égaux à *x*.

Nouveau dans la version 3.2.

extend(iterable)

Étend la *deque* en ajoutant les éléments de l'itérable en argument à son extrémité droite.

extendleft(iterable)

Étend la *deque* en ajoutant les éléments d'*iterable* à son extrémité gauche. Dans ce cas, notez que la série d'ajouts inverse l'ordre des éléments de l'argument itérable.

index(x[, start[, stop]])

Renvoie la position de *x* dans la *deque* (à partir de *start* inclus et jusqu'à *stop* exclus). Renvoie la première correspondance ou lève *ValueError* si aucune n'est trouvée.

Nouveau dans la version 3.5.

insert(i, x)

Insère *x* dans la *deque* à la position *i*.

Si une insertion provoque un dépassement de la taille limitée d'une *deque*, alors elle lève une exception *IndexError*.

Nouveau dans la version 3.5.

pop()

Retire et renvoie un élément de l'extrémité droite de la *deque*. S'il n'y a aucun élément, lève une exception *IndexError*.

popleft()

Retire et renvoie un élément de l'extrémité gauche de la *deque*. S'il n'y a aucun élément, lève une exception *IndexError*.

remove(value)

Supprime la première occurrence de *value*. Si aucune occurrence n'est trouvée, lève une exception *ValueError*.

reverse()

Inverse le sens des éléments de la *deque* sans créer de copie et renvoie *None*.

Nouveau dans la version 3.2.

rotate(n=1)

Décale les éléments de la *deque* de *n* places vers la droite (le dernier élément revient au début). Si *n* est négatif, décale vers la gauche.

Quand la *deque* n'est pas vide, un décalage d'une place vers la droite équivaut à `d.appendleft(d.pop())` et un décalage d'une place vers la gauche est équivalent à `d.append(d.popleft())`.

Les objets *deques* fournissent également un attribut en lecture seule :

maxlen

La taille maximale d'une *deque*, ou *None* si illimitée.

Nouveau dans la version 3.1.

En plus des méthodes précédentes, les *deques* gèrent l'itération, la sérialisation, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, le test d'appartenance avec l'opérateur `in`, et les références en indice comme `d[-1]`. L'accès par indice est en $O(1)$ aux extrémités mais en $O(n)$ au milieu. Pour des accès aléatoires rapides, il est préférable d'utiliser des listes.

Depuis la version 3.5, les *deques* gèrent `__add__()`, `__mul__()` et `__imul__()`.

Exemple :

```

>>> from collections import deque
>>> d = deque('ghi')           # make a new deque with three items
>>> for elem in d:             # iterate over the deque's elements
...     print(elem.upper())
G
H
I

>>> d.append('j')              # add a new entry to the right side
>>> d.appendleft('f')          # add a new entry to the left side
>>> d                           # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                    # return and remove the rightmost item
'j'
>>> d.popleft()                # return and remove the leftmost item
'f'
>>> list(d)                    # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                       # peek at leftmost item
'g'
>>> d[-1]                     # peek at rightmost item
'i'

>>> list(reversed(d))          # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                   # search the deque
True
>>> d.extend('jkl')            # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                 # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))          # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                   # empty the deque
>>> d.pop()                     # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')         # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])

```

Cas pratiques utilisant deque

Cette partie montre diverses approches afin de travailler avec les *deques*.

Les *deques* à taille limitée apportent une fonctionnalité similaire au filtre `tail` d'Unix :

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Une autre approche d'utilisation des *deques* est de maintenir une séquence d'éléments récemment ajoutés en les ajoutant à droite et en retirant les anciens par la gauche :

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

The `rotate()` method provides a way to implement *deque* slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped :

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement *deque* slicing, use a similar approach applying `rotate()` to bring a target element to the left side of the deque. Remove old entries with `popleft()`, add new entries with `extend()`, and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as `dup`, `drop`, `swap`, `over`, `pick`, `rot`, and `roll`.

8.3.4 Objets defaultdict

class `collections.defaultdict` (`[default_factory[, ...]]`)

Renvoie un nouvel objet qui se comporte comme un dictionnaire. *defaultdict* est une sous-classe de la classe native *dict*. Elle surcharge une méthode et ajoute une variable d'instance modifiable. Les autres fonctionnalités sont les mêmes que celles des objets *dict* et ne sont pas documentées ici.

Le premier argument fournit la valeur initiale de l'attribut *default_factory* qui doit être un objet callable sans paramètre ou `None`, sa valeur par défaut. Tous les autres arguments sont traités comme si on les passait au constructeur de *dict*, y compris les arguments nommés.

En plus des opérations usuelles de *dict*, les objets *defaultdict* gèrent les méthodes supplémentaires suivantes :

`__missing__` (*key*)

Si l'attribut *default_factory* est `None`, lève une exception *KeyError* avec *key* comme argument.

Si *default_factory* ne vaut pas `None`, cet attribut est appelé sans argument pour fournir une valeur par défaut pour la *key* demandée. Cette valeur est insérée dans le dictionnaire avec pour clé *key* et est renvoyée.

Si appeler *default_factory* lève une exception, celle-ci est transmise inchangée.

Cette méthode est appelée par la méthode `__getitem__()` de la classe `dict` lorsque la clé demandée n'est pas trouvée. Ce qu'elle renvoie ou lève est alors renvoyé ou levé par `__getitem__()`.

Remarquez que `__missing__()` n'est *pas* appelée pour les opérations autres que `__getitem__()`. Cela signifie que `get()` renvoie `None` comme les dictionnaires natifs dans les cas triviaux et n'utilise pas `default_factory`.

Les objets `defaultdict` gèrent la variable d'instance :

default_factory

Cet attribut est utilisé par la méthode `__missing__()` ; il est initialisé par le premier argument passé au constructeur, s'il est spécifié, sinon par `None`.

Exemples utilisant defaultdict

Using `list` as the `default_factory`, it is easy to group a sequence of key-value pairs into a dictionary of lists :

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping ; so an entry is automatically created using the `default_factory` function which returns an empty `list`. The `list.append()` operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the `list.append()` operation adds another value to the list. This technique is simpler and faster than an equivalent technique using `dict.setdefault()` :

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the `default_factory` to `int` makes the `defaultdict` useful for counting (like a bag or multiset in other languages) :

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

When a letter is first encountered, it is missing from the mapping, so the `default_factory` function calls `int()` to supply a default count of zero. The increment operation then builds up the count for each letter.

La fonction `int()` qui retourne toujours zéro est simplement une fonction constante particulière. Un moyen plus flexible et rapide de créer une fonction constante est d'utiliser une fonction lambda qui peut fournir n'importe quelle valeur constante (pas seulement zéro) :

```
>>> def constant_factory(value):
...     return lambda: value
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Setting the `default_factory` to `set` makes the `defaultdict` useful for building a dictionary of sets :

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.3.5 `namedtuple()` : fonction de construction pour *n-uplets* (*tuples*) avec des champs nommés

Les tuples nommés assignent une signification à chacun de leur élément, ce qui rend le code plus lisible et explicite. Ils peuvent être utilisés partout où les tuples natifs sont utilisés, et ils ajoutent la possibilité d'accéder à leurs champs grâce à leur nom au lieu de leur index de position.

`collections.namedtuple` (*typename*, *field_names*, *, *verbose=False*, *rename=False*, *module=None*)

Renvoie une nouvelle sous-classe de `tuple` appelée *typename*. Elle est utilisée pour créer des objets se comportant comme les *tuples* qui ont des champs accessibles par recherche d'attribut en plus d'être indexables et itérables. Les instances de cette sous-classe possèdent aussi une *docstring* explicite (avec *type_name* et les *field_names*) et une méthode `__repr__()` pratique qui liste le contenu du tuple au format `nom=valeur`.

field_names peut être une séquence de chaînes de caractères telle que `['x', 'y']` ou bien une unique chaîne de caractères où les noms de champs sont séparés par un espace et/ou une virgule, par exemple `'x y'` ou `'x, y'`. N'importe quel identifiant Python peut être utilisé pour un nom de champ hormis ceux commençant par un tiret bas. Les identifiants valides peuvent contenir des lettres, des chiffres (sauf en première position) et des tirets bas (sauf en première position). Un identifiant ne peut pas être un *keyword* tel que `class`, `for`, `return`, `global`, `pass` ou `raise`.

Si *rename* vaut `True`, alors les noms de champs invalides sont automatiquement renommés en noms positionnels. Par exemple, `['abc', 'def', 'ghi', 'abc']` est converti en `['abc', '_1', 'ghi', '_3']` afin d'éliminer le mot-clé `def` et le doublon de `abc`.

If *verbose* is true, the class definition is printed after it is built. This option is outdated ; instead, it is simpler to print the `__source__` attribute.

Si *module* est spécifié, alors il est assigné à l'attribut `__module__` du tuple nommé.

Les instances de tuples nommés n'ont pas de dictionnaires propres, elles sont donc légères et ne requièrent pas plus de mémoire que les tuples natifs.

Modifié dans la version 3.1 : Gestion de *rename*.

Modifié dans la version 3.6 : Les paramètres *verbose* et *rename* deviennent des *arguments obligatoirement nommés*.

Modifié dans la version 3.6 : Ajout du paramètre *module*.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)           # instantiate with positional or keyword arguments
>>> p[0] + p[1]                   # indexable like the plain tuple (11, 22)
33
>>> x, y = p                      # unpack like a regular tuple
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> x, y
(11, 22)
>>> p.x + p.y           # fields also accessible by name
33
>>> p                   # readable __repr__ with a name=value style
Point(x=11, y=22)
```

Les tuples nommés sont particulièrement utiles pour associer des noms de champs à des tuples renvoyés par les modules *csv* ou *sqlite3*:

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade
↪')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

En plus des méthodes héritées de `tuple`, les tuples nommés implémentent trois méthodes et deux attributs supplémentaires. Pour éviter les conflits avec noms de champs, leurs noms commencent par un tiret bas.

classmethod `somenamedtuple._make(iterable)`

Méthode de classe qui construit une nouvelle instance à partir d'une séquence ou d'un itérable existant.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

`somenamedtuple._asdict()`

Renvoie un nouvel *OrderedDict* qui associe chaque nom de champ à sa valeur correspondante :

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

Modifié dans la version 3.1 : Renvoie un *OrderedDict* au lieu d'un *dict* natif.

`somenamedtuple._replace(**kwargs)`

Renvoie une nouvelle instance du tuple nommé en remplaçant les champs spécifiés par leurs nouvelles valeurs :

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum], ↪
↪timestamp=time.now())
```

`somenamedtuple._source`

A string with the pure Python source code used to create the named tuple class. The source makes the named tuple self-documenting. It can be printed, executed using *exec()*, or saved to a file and imported.

Nouveau dans la version 3.3.

`somenamedtuple._fields`

Tuple de chaînes de caractères listant les noms de champs. Pratique pour l'inspection et pour créer de nouveaux types de tuples nommés à partir d'existants.

```
>>> p._fields          # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

Pour récupérer un champ dont le nom est une chaîne de caractères, utilisez la fonction `getattr()` :

```
>>> getattr(p, 'x')
11
```

Pour convertir un dictionnaire en tuple nommé, utilisez l'opérateur double-étoile (comme expliqué dans `tut-unpacking-arguments`) :

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Il est aisé d'ajouter ou de modifier les fonctionnalités des tuples nommés grâce à l'héritage puisqu'il s'agit de simples classes. Voici comment ajouter un champ calculé avec une longueur fixe d'affichage :

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f y=%6.3f hypot=%6.3f' % (self.x, self.y, self.
↳ hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000 y= 4.000 hypot= 5.000
Point: x=14.000 y= 0.714 hypot=14.018
```

La sous-classe ci-dessus définit `__slots__` comme un tuple vide. Cela permet de garder une empreinte mémoire faible en empêchant la création de dictionnaire d'instance.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the `_fields` attribute :

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Les `docstrings` peuvent être personnalisées en modifiant directement l'attribut `__doc__` :

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

Modifié dans la version 3.5 : La propriété devient éditable.

Default values can be implemented by using `_replace()` to customize a prototype instance :

```
>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
>>> janes_account = default_account._replace(owner='Jane')
```

Voir aussi :

- Cas pratique : classe de base abstraite de tuple nommé grâce à une métaclass `mixin` par Jan Kaliszewski. En plus de fournir une *abstract base class* pour les tuples nommés, elle gère un constructeur de *metaclass* pratique dans les cas où l'on hérite de tuples nommés.
- Voir `types.SimpleNamespace()` pour un espace de nommage muable basé sur un dictionnaire sous-jacent à la place d'un tuple.
- Voir `typing.NamedTuple()` pour un moyen d'ajouter des indications de type pour les tuples nommés.

8.3.6 Objets `OrderedDict`

En plus de se comporter comme des dictionnaires natifs, les dictionnaires ordonnés mémorisent l'ordre dans lequel les éléments ont été insérés. Quand on itère sur un dictionnaire ordonné, les éléments sont renvoyés dans l'ordre d'insertion des clés.

class `collections.OrderedDict` (`[items]`)

Renvoie une instance d'une sous-classe de `dict` qui gère les méthodes usuelles de `dict`. Un objet `OrderedDict` est un dictionnaire qui mémorise l'ordre d'insertion des clés. Si une nouvelle entrée écrase une autre, sa position reste inchangé. Si une entrée est supprimée puis réinsérée, elle est placée en dernière position.

Nouveau dans la version 3.1.

popitem (`last=True`)

La méthode `popitem()` pour les dictionnaires ordonnés retire et renvoie une paire (clé, valeur). Les paires sont renvoyées comme pour une pile, c'est-à-dire dernier entré, premier sorti (en anglais LIFO (last-in, first-out)) si `last` vaut `True`. Si `last` vaut `False`, alors les paires sont renvoyées comme pour une file, c'est-à-dire premier entré, premier sorti (en anglais FIFO (first-in, first-out)).

move_to_end (`key`, `last=True`)

Déplace une clé `key` existante à l'une des deux extrémités du dictionnaire : à droite si `last` vaut `True` (comportement par défaut) ou à gauche sinon. Lève une exception `KeyError` si la clé `key` n'est pas trouvée :

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

Nouveau dans la version 3.2.

En plus des méthodes usuelles des dictionnaires, les dictionnaires ordonnés gèrent l'itération en sens inverse grâce à `reversed()`.

Les tests d'égalité entre deux objets `OrderedDict` sont sensibles à l'ordre et sont implémentés comme ceci : `list(od1.items()) == list(od2.items())`. Les tests d'égalité entre un objet `OrderedDict` et un objet `Mapping` ne sont pas sensibles à l'ordre (comme les dictionnaires natifs). Cela permet substituer des objets `OrderedDict` partout où les dictionnaires natifs sont utilisés.

Modifié dans la version 3.5 : Les *vues* d'éléments, de clés et de valeurs de `OrderedDict` gèrent maintenant l'itération en sens inverse en utilisant `reversed()`.

Modifié dans la version 3.6 : Suite à l'acceptation de la [PEP 468](#), l'ordre des arguments nommés passés au constructeur et à la méthode `update()` de `OrderedDict` est conservé.

Exemples et cas pratiques utilisant `OrderedDict`

Puisqu'un dictionnaire ordonné mémorise l'ordre d'insertion de ses éléments, il peut être utilisé conjointement avec un classement pour créer un dictionnaire trié :

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

Les nouveaux dictionnaires triés gardent leur classement quand des entrées sont supprimées, mais si de nouvelles clés sont ajoutées, celles-ci sont ajoutée à la fin et le classement est perdu.

Il est également facile de créer une variante de dictionnaire ordonné qui retient l'ordre dans lequel les clés ont été insérées *en dernier*. Si une nouvelle entrée écrase une existante, la position d'insertion d'origine est modifiée et déplacée à la fin :

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        OrderedDict.__setitem__(self, key, value)
```

Un dictionnaire ordonné peut être combiné avec la classe `Counter` afin de mémoriser l'ordre dans lequel les éléments ont été ajoutés pour la première fois :

```
class OrderedCounter(Counter, OrderedDict):
    'Counter that remembers the order elements are first encountered'

    def __repr__(self):
        return '%s(%r)' % (self.__class__.__name__, OrderedDict(self))

    def __reduce__(self):
        return self.__class__, (OrderedDict(self),)
```

8.3.7 Objets `UserDict`

La classe `UserDict` se comporte comme une surcouche autour des dictionnaires. L'utilité de cette classe est réduite car on peut maintenant hériter directement de `dict`. Cependant, il peut être plus facile de travailler avec celle-ci car le dictionnaire sous-jacent est accessible comme attribut.

class `collections.UserDict` (`[initialdata]`)

Classe simulant un dictionnaire. Les instances de `UserDict` possèdent un attribut `data` où est stocké leur contenu sous forme de dictionnaire natif. Si `initialdata` est spécifié, alors `data` est initialisé avec son contenu. Remarquez qu'une référence vers `initialdata` n'est pas conservée, ce qui permet de l'utiliser pour d'autres tâches.

En plus de gérer les méthodes et opérations des dictionnaires, les instances de `UserDict` fournissent l'attribut suivant :

data

Un dictionnaire natif où est stocké le contenu de la classe `UserDict`.

8.3.8 Objets `UserList`

Cette classe agit comme une surcouche autour des objets `list`. C'est une classe mère utile pour vos classes listes-compatibles qui peuvent en hériter et surcharger les méthodes existantes ou en ajouter de nouvelles. Ainsi, on peut ajouter de nouveaux comportements aux listes.

L'utilité de cette classe a été partiellement réduite par la possibilité d'hériter directement de `list`. Cependant, il peut être plus facile de travailler avec cette classe car la liste sous-jacente est accessible via un attribut.

class `collections.UserList` (`[list]`)

Classe simulant une liste. Les instances de `UserList` possèdent un attribut `UserList` où est stocké leur contenu sous forme de liste native. Il est initialement une copie de `list`, ou `[]` par défaut. `list` peut être un itérable, par exemple une liste native ou un objet `UserList`.

En plus de gérer les méthodes et opérations des séquences muables, les instances de `UserList` possèdent l'attribut suivant :

data

Un objet `list` natif utilisé pour stocker le contenu de la classe `UserList`.

Prérequis pour l'héritage : Les sous-classes de `UserList` doivent implémenter un constructeur qui peut être appelé avec zéro ou un argument. Les opérations sur les listes qui renvoient une nouvelle séquence essaient de créer une instance de la classe courante. C'est pour cela que le constructeur doit pouvoir être appelé avec un unique paramètre, un objet séquence utilisé comme source de données.

Si une classe fille ne remplit pas cette condition, toutes les méthodes spéciales gérées par cette classe devront être implémentées à nouveau. Merci de consulter les sources pour obtenir des informations sur les méthodes qui doivent être fournies dans ce cas.

8.3.9 Objets `UserString`

La classe `UserString` agit comme une surcouche autour des objets `str`. L'utilité de cette classe a été partiellement réduite par la possibilité d'hériter directement de `str`. Cependant, il peut être plus facile de travailler avec cette classe car la chaîne de caractère sous-jacente est accessible via un attribut.

class `collections.UserString` (`[sequence]`)

Class that simulates a string or a Unicode string object. The instance's content is kept in a regular string object, which is accessible via the `data` attribute of `UserString` instances. The instance's contents are initially set to a copy of `sequence`. The `sequence` can be an instance of `bytes`, `str`, `UserString` (or a subclass) or an arbitrary sequence which can be converted into a string using the built-in `str()` function.

Modifié dans la version 3.5 : Nouvelles méthodes `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable` et `maketrans`.

8.4 `collections.abc` — Classes de base abstraites pour les conteneurs

Nouveau dans la version 3.3 : Auparavant, ce module faisait partie du module `collections`.

Code source : [Lib/_collections_abc.py](#)

Ce module fournit *des classes de base abstraites* qui peuvent être utilisées pour vérifier si une classe fournit une interface particulière (par exemple, savoir s’il s’agit d’un hachable ou d’une table de correspondance).

8.4.1 Classes de base abstraites de collections

Le module `collections` apporte les *ABC* suivantes :

ABC	Hérite de	Méthodes abstraites	Méthodes <i>mixin</i>
<i>Container</i>		<code>__contains__</code>	
<i>Hashable</i>		<code>__hash__</code>	
<i>Iterable</i>		<code>__iter__</code>	
<i>Iterator</i>	<i>Iterable</i>	<code>__next__</code>	<code>__iter__</code>
<i>Reversible</i>	<i>Iterable</i>	<code>__reversed__</code>	
<i>Generator</i>	<i>Iterator</i>	<code>send</code> , <code>throw</code>	<code>close</code> , <code>__iter__</code> , <code>__next__</code>
<i>Sized</i>		<code>__len__</code>	
<i>Callable</i>		<code>__call__</code>	
<i>Collection</i>	<i>Sized</i> , <i>Iterable</i> , <i>Container</i>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>	
<i>Sequence</i>	<i>Reversible</i> , <i>Collection</i>	<code>__getitem__</code> , <code>__len__</code>	<code>__contains__</code> , <code>__iter__</code> , <code>__reversed__</code> , <code>index</code> et <code>count</code>
<i>MutableSequence</i>	<i>Sequence</i>	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__len__</code> , <code>insert</code>	Méthodes héritées de <i>Sequence</i> , et <code>append</code> , <code>reverse</code> , <code>extend</code> , <code>pop</code> , <code>remove</code> et <code>__iadd__</code>
<i>ByteString</i>	<i>Sequence</i>	<code>__getitem__</code> , <code>__len__</code>	Méthodes héritées de <i>Sequence</i>
<i>Set</i>	<i>Collection</i>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>	<code>__le__</code> , <code>__lt__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code> , <code>__and__</code> , <code>__or__</code> , <code>__sub__</code> , <code>__xor__</code> et <code>isdisjoint</code>
<i>MutableSet</i>	<i>Set</i>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code> , <code>add</code> , <code>discard</code>	Méthodes héritées de <i>Set</i> , et <code>clear</code> , <code>pop</code> , <code>remove</code> , <code>__ior__</code> , <code>__iand__</code> , <code>__ixor__</code> et <code>__isub__</code>
<i>Mapping</i>	<i>Collection</i>	<code>__getitem__</code> , <code>__iter__</code> , <code>__len__</code>	<code>__contains__</code> , <code>keys</code> , <code>items</code> , <code>values</code> , <code>get</code> , <code>__eq__</code> et <code>__ne__</code>
<i>MutableMapping</i>	<i>Mapping</i>	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__iter__</code> , <code>__len__</code>	Méthodes héritées de <i>Mapping</i> , et <code>pop</code> , <code>popitem</code> , <code>clear</code> , <code>update</code> et <code>setdefault</code>
<i>MappingView</i>	<i>Sized</i>		<code>__len__</code>
<i>ItemsView</i>	<i>MappingView</i> , <i>Set</i>		<code>__contains__</code> , <code>__iter__</code>
<i>KeysView</i>	<i>MappingView</i> , <i>Set</i>		<code>__contains__</code> , <code>__iter__</code>
<i>ValuesView</i>	<i>MappingView</i>		<code>__contains__</code> , <code>__iter__</code>
<i>Awaitable</i>		<code>__await__</code>	
<i>Coroutine</i>	<i>Awaitable</i>	<code>send</code> , <code>throw</code>	<code>close</code>
<i>AsyncIterable</i>		<code>__aiter__</code>	
<i>AsyncIterator</i>	<i>AsyncIterable</i>	<code>__anext__</code>	<code>__aiter__</code>
<i>AsyncGenerator</i>	<i>AsyncIterator</i>	<code>asend</code> , <code>athrow</code>	<code>aclose</code> , <code>__aiter__</code> , <code>__anext__</code>

```
class collections.abc.Container
```

```
class collections.abc.Hashable
```

```
class collections.abc.Sized
```

```
class collections.abc.Callable
```

ABC pour les classes qui définissent respectivement les méthodes `__contains__()`, `__hash__()`, `__len__()` et `__call__()`.

```
class collections.abc.Iterable
```

ABC pour les classes qui définissent la méthode `__iter__()`.

Évaluer `isinstance(obj, Iterable)` détecte les classes qui sont enregistrées comme *Iterable* ou qui possèdent une méthode `__iter__()`, mais ne détecte pas les classes qui itèrent avec la méthode `__getitem__()`. Le seul moyen fiable de déterminer si un objet est *itérable* est d'appeler `iter(obj)`.

class `collections.abc.Collection`

ABC pour les classes de conteneurs itérables et *sized*.

Nouveau dans la version 3.6.

class `collections.abc.Iterator`

ABC pour les classes qui définissent les méthodes `__iter__()` et `__next__()`. Voir aussi la définition d'*itérateur*.

class `collections.abc.Reversible`

ABC pour les classes d'itérables qui implémentent également la méthode `__reversed__()`.

Nouveau dans la version 3.6.

class `collections.abc.Generator`

ABC pour les classes de générateurs qui implémentent le protocole défini dans la **PEP 342** qui étend les itérateurs avec les méthodes `send()`, `throw()` et `close()`. Voir aussi la définition de *générateur*.

Nouveau dans la version 3.5.

class `collections.abc.Sequence`

class `collections.abc.MutableSequence`

class `collections.abc.ByteString`

ABC pour les *séquences* immuables et muables.

Note pour l'implémentation : quelques méthodes *mixin*, comme `__iter__()`, `__reversed__()` et `index()`, font des appels répétés à la méthode sous-jacente `__getitem__()`. Ainsi, si `__getitem__()` est implémentée avec une vitesse d'accès constante, les méthodes *mixin* auront une performance linéaire ; cependant, si elle est linéaire, les *mixin* auront une performance quadratique, il serait alors judicieux de les surcharger.

Modifié dans la version 3.5 : La méthode `index()` a ajouté le support des arguments *start* et *stop*.

class `collections.abc.Set`

class `collections.abc.MutableSet`

ABC pour les ensembles immuables et muables.

class `collections.abc.Mapping`

class `collections.abc.MutableMapping`

ABC pour les *tables de correspondances* immuables et muables.

class `collections.abc.MappingView`

class `collections.abc.ItemsView`

class `collections.abc.KeysView`

class `collections.abc.ValuesView`

ABC pour les *vues de mappings* (tableaux de correspondances), d'éléments, de clés et de valeurs.

class `collections.abc.Awaitable`

ABC pour les objets *awaitables*, qui peuvent être utilisés dans les expressions `await`. Les implémentations personnalisées doivent définir la méthode `__await__()`.

Les objets *coroutines* et les instances de l'ABC *Coroutine* sont tous des instances de cette ABC.

Note : En CPython, les coroutines basées sur les générateurs (les générateurs décorés avec `types.coroutine()` ou `asyncio.coroutine()`) sont *awaitables*, bien qu'elles n'aient pas de méthode `__await__()`. Évaluer `isinstance(gencoro, Awaitable)` où `gencoro` est un générateur décoré va renvoyer `False`. Utilisez `inspect.isawaitable()` pour les détecter.

Nouveau dans la version 3.5.

class `collections.abc.Coroutine`

ABC pour les classes compatibles avec les coroutines. Elles implémentent les méthodes suivantes, définies dans `coroutine-objects` : `send()`, `throw()` et `close()`. Les implémentations personnalisées doivent également fournir `__await__()`. Toutes les instances de *Coroutine* sont également des instances de *Awaitable*. Voir aussi la définition de *coroutine*.

Note : En CPython, les coroutines basées sur les générateurs (les générateurs décorés avec `types.coroutine()` ou `asyncio.coroutine()`) sont *awaitables*, bien qu'elles n'aient pas de méthode `__await__()`. Évaluer `isinstance(gencoro, Coroutine)` où `gencoro` est un générateur décoré va renvoyer `False`. Utilisez `inspect.isawaitable()` pour les détecter.

Nouveau dans la version 3.5.

class `collections.abc.AsyncIterable`

ABC pour les classes qui définissent la méthode `__aiter__`. Voir aussi la définition d'*itérable asynchrone*.

Nouveau dans la version 3.5.

class `collections.abc.AsyncIterator`

ABC pour les classes qui définissent les méthodes `__aiter__` et `__anext__`. Voir aussi la définition d'*itérateur asynchrone*.

Nouveau dans la version 3.5.

class `collections.abc.AsyncGenerator`

ABC pour les classes de générateurs asynchrones qui implémentent le protocole défini dans la **PEP 525** et dans la **PEP 492**.

Nouveau dans la version 3.6.

Ces ABC permettent de demander à des classes ou à des instances si elles fournissent des fonctionnalités particulières, par exemple :

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

Une partie des ABC sont également utiles en tant que *mixins* : cela rend plus facile le développement de classes qui gèrent des API de conteneurs. Par exemple, pour écrire une classe qui gère l'API entière de *Set*, il est uniquement nécessaire de fournir les trois méthodes sous-jacentes abstraites `__contains__()`, `__iter__()` et `__len__()`. L'ABC apporte les méthodes restantes, comme `__and__()` et `isdisjoint()` :

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
    and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)
```

(suite sur la page suivante)

(suite de la page précédente)

```
s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2           # The __and__() method is supported automatically
```

Notes à propos de l'utilisation de *Set* et *MutableSet* comme *mixin* :

- (1) Comme une partie des opérations sur les ensembles créent de nouveaux ensembles, les méthodes *mixins* par défaut ont besoin d'un moyen de créer de nouvelles instances à partir d'un itérable. Le constructeur de classe est supposé avoir une signature de la forme `ClassName(iterable)`. Cette supposition est faite par une méthode de classe interne appelée `_from_iterable()` qui appelle `cls(iterable)` pour construire un nouvel ensemble. Si le *Set* *mixin* est utilisé dans une classe avec un constructeur de signature différente, vous devrez surcharger `_from_iterable()` avec une méthode de classe qui peut construire de nouvelles instances à partir d'un argument itérable.
- (2) Pour surcharger les comparaisons (a priori pour la rapidité, puisque la sémantique est fixe), il faut redéfinir `__le__()` et `__ge__()`, puis les autres opérations seront automatiquement adaptées.
- (3) La classe *mixin* *Set* apporte une méthode `__hash__()` pour calculer une valeur de hachage pour l'ensemble ; cependant `__hash__()` n'est pas défini car tous les ensembles ne sont pas hachables ou immuables. Pour rendre un ensemble hachable en utilisant les *mixins*, héritez de *Set()* et de *Hashable()*, puis définissez `__hash__ = Set.__hash__`.

Voir aussi :

- [OrderedSet](#) *recipe* pour un exemple construit sur *MutableSet*.
- Pour plus d'informations à propos des ABC, voir le module *abc* et la [PEP 3119](#).

8.5 heapq — File de priorité basée sur un tas

Code source : [Lib/heapq.py](#)

Ce module expose une implémentation de l'algorithme de file de priorité, basée sur un tas.

Les tas sont des arbres binaires pour lesquels chaque valeur portée par un nœud est inférieure ou égale à celle de ses deux fils. Cette implémentation utilise des tableaux pour lesquels `tas[k] <= tas[2*k+1]` et `tas[k] <= tas[2*k+2]` pour tout *k*, en commençant la numérotation à zéro. Pour contenter l'opérateur de comparaison, les éléments inexistants sont considérés comme porteur d'une valeur infinie. L'intérêt du tas est que son plus petit élément est toujours la racine, `tas[0]`.

L'API ci-dessous diffère de la file de priorité classique par deux aspects : (a) L'indilage commence à zéro. Cela complexifie légèrement la relation entre l'indice d'un nœud et les indices de ses fils mais est alignée avec l'indilage commençant à zéro que Python utilise. (b) La méthode *pop* renvoie le plus petit élément et non le plus grand (appelé « tas-min » dans les manuels scolaires ; le « tas-max » étant généralement plus courant dans la littérature car il permet le classement sans tampon).

Ces deux points permettent d'aborder le tas comme une liste Python standard sans surprise : `heap[0]` est le plus petit élément tandis que `heap.sort()` ne modifie pas le tas !

Pour créer un tas, utilisez une liste initialisée à `[]` ou bien utilisez une liste existante et transformez la en tas à l'aide de la fonction `heapify()`.

Les fonctions suivantes sont fournies :

`heapq.heappush(heap, item)`

Introduit la valeur *item* dans le tas *heap*, en conservant l'invariance du tas.

`heapq.heappop(heap)`

Extraie le plus petit élément de *heap* en préservant l'invariant du tas. Si le tas est vide, une exception `IndexError` est levée. Pour accéder au plus petit élément sans le retirer, utilisez `heap[0]`.

`heapq.heappushpop(heap, item)`

Introduit l'élément *item* dans le tas, puis extraie le plus petit élément de *heap*. Cette action combinée est plus efficace que `heappush()` suivie par un appel séparé à `heappop()`.

`heapq.heapify(x)`

Transforme une liste *x* en un tas, sans utiliser de tampon et en temps linéaire.

`heapq.heapreplace(heap, item)`

Extraie le plus petit élément de *heap* et introduit le nouvel élément *item*. La taille du tas ne change pas. Si le tas est vide, une exception `IndexError` est levée.

Cette opération en une étape est plus efficace qu'un appel à `heappop()` suivi d'un appel à `heappush()` et est plus appropriée lorsque le tas est de taille fixe. La combinaison *pop/push* renvoie toujours un élément du tas et le remplace par *item*.

La valeur renvoyée peut être plus grande que l'élément *item* ajouté. Si cela n'est pas souhaitable, utilisez plutôt `heappushpop()` à la place. Sa combinaison *push/pop* renvoie le plus petit élément des deux valeurs et laisse la plus grande sur le tas.

Ce module contient également trois fonctions génériques utilisant les tas.

`heapq.merge(*iterables, key=None, reverse=False)`

Fusionne plusieurs entrées ordonnées en une unique sortie ordonnée (par exemple, fusionne des entrées datées provenant de multiples journaux applicatifs). Renvoie un `iterator` sur les valeurs ordonnées.

Similaire à `sorted(itertools.chain(*iterables))` mais renvoie un itérable, ne stocke pas toutes les données en mémoire en une fois et suppose que chaque flux d'entrée est déjà classé (en ordre croissant).

A deux arguments optionnels qui doivent être fournis par mot clef.

key spécifie une *key function* d'un argument utilisée pour extraire une clef de comparaison de chaque élément de la liste. La valeur par défaut est `None` (compare les éléments directement).

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

Modifié dans la version 3.5 : Ajout des paramètres optionnels *key* et *reverse*.

`heapq.nlargest(n, iterable, key=None)`

Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to `:sorted(iterable, key=key, reverse=True)[:n]`.

`heapq.nsmallest(n, iterable, key=None)`

Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to `:sorted(iterable, key=key)[:n]`.

Les deux fonctions précédentes sont les plus efficaces pour des petites valeurs de *n*. Pour de grandes valeurs, il est préférable d'utiliser la fonction `sorted()`. En outre, lorsque *n*=1, il est plus efficace d'utiliser les fonctions natives `min()` et `max()`. Si vous devez utiliser ces fonctions de façon répétée, il est préférable de transformer l'itérable en tas.

8.5.1 Exemples simples

Un **tri par tas** peut être implémenté en introduisant toutes les valeurs dans un tas puis en effectuant l'extraction des éléments un par un :

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ceci est similaire à `sorted(iterable)` mais, contrairement à `sorted()`, cette implémentation n'est pas stable.

Les éléments d'un tas peuvent être des n-uplets. C'est pratique pour assigner des valeurs de comparaison (par exemple, des priorités de tâches) en plus de l'élément qui est suivi :

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.5.2 Notes d'implémentation de la file de priorité

Une **file de priorité** est une application courante des tas et présente plusieurs défis d'implémentation :

- Stabilité du classement : comment s'assurer que deux tâches avec la même priorité sont renvoyées dans l'ordre de leur ajout ?
- La comparaison des couples (priorité, tâche) échoue si les priorités sont identiques et que les tâches n'ont pas de relation d'ordre par défaut.
- Si la priorité d'une tâche change, comment la déplacer à sa nouvelle position dans le tas ?
- Si une tâche en attente doit être supprimée, comment la trouver et la supprimer de la file ?

Une solution aux deux premiers problèmes consiste à stocker les entrées sous forme de liste à 3 éléments incluant la priorité, le numéro d'ajout et la tâche. Le numéro d'ajout sert à briser les égalités de telle sorte que deux tâches avec la même priorité sont renvoyées dans l'ordre de leur insertion. Puisque deux tâches ne peuvent jamais avoir le même numéro d'ajout, la comparaison des triplets ne va jamais chercher à comparer des tâches entre elles.

Le problème restant consiste à trouver une tâche en attente et modifier sa priorité ou la supprimer. Trouver une tâche peut être réalisé à l'aide d'un dictionnaire pointant vers une entrée dans la file.

Supprimer une entrée ou changer sa priorité est plus difficile puisque cela romprait l'invariant de la structure de tas. Une solution possible est de marquer l'entrée comme supprimée et d'ajouter une nouvelle entrée avec sa priorité modifiée :

```
pq = []                                # list of entries arranged in a heap
entry_finder = {}                       # mapping of tasks to entries
REMOVED = '<removed-task>'              # placeholder for a removed task
counter = itertools.count()             # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
```

(suite sur la page suivante)

(suite de la page précédente)

```

count = next(counter)
entry = [priority, count, task]
entry_finder[task] = entry
heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED. Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

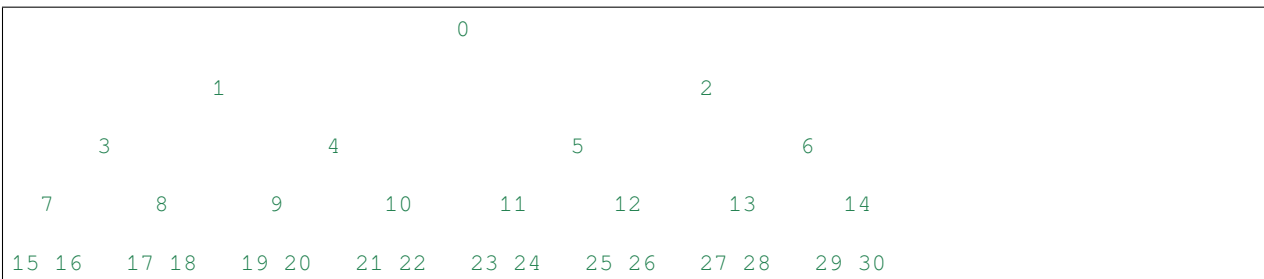
def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')

```

8.5.3 Théorie

Les tas sont des tableaux pour lesquels $a[k] \leq a[2k+1]$ et $a[k] \leq a[2k+2]$ pour tout k en comptant les éléments à partir de 0. Pour simplifier la comparaison, les éléments inexistants sont considérés comme étant infinis. L'intérêt des tas est que $a[0]$ est toujours leur plus petit élément.

L'invariant étrange ci-dessus est une représentation efficace en mémoire d'un tournoi. Les nombres ci-dessous sont k et non $a[k]$:



Dans l'arbre ci-dessus, chaque nœud k a pour enfants $2k+1$ et $2k+2$. Dans les tournois binaires habituels dans les compétitions sportives, chaque nœud est le vainqueur des deux nœuds inférieurs et nous pouvons tracer le chemin du vainqueur le long de l'arbre afin de voir qui étaient ses adversaires. Cependant, dans de nombreuses applications informatiques de ces tournois, nous n'avons pas besoin de produire l'historique du vainqueur. Afin d'occuper moins de mémoire, on remplace le vainqueur lors de sa promotion par un autre élément à un plus bas niveau. La règle devient alors qu'un nœud et les deux nœuds qu'il chapeaute contiennent trois éléments différents, mais le nœud supérieur « gagne » contre les deux nœuds inférieurs.

Si cet invariant de tas est vérifié à tout instant, alors l'élément à l'indice 0 est le vainqueur global. L'algorithme le plus simple pour le retirer et trouver le vainqueur « suivant » consiste à déplacer un perdant (par exemple le nœud 30 dans le diagramme ci-dessus) à la position 0, puis à faire redescendre cette nouvelle racine dans l'arbre en échangeant sa valeur avec celle d'un de ses fils jusqu'à ce que l'invariant soit rétabli. Cette approche a un coût logarithmique par rapport au nombre total d'éléments dans l'arbre. En itérant sur tous les éléments, le classement s'effectue en $O(n \log n)$ opérations.

Une propriété agréable de cet algorithme est qu'il est possible d'insérer efficacement de nouveaux éléments en cours de classement, du moment que les éléments insérés ne sont pas « meilleurs » que le dernier élément qui a été extrait. Ceci s'avère très utile dans des simulations où l'arbre contient la liste des événements arrivants et que la condition de « victoire

» est le plus petit temps d'exécution planifié. Lorsqu'un événement programme l'exécution d'autres événements, ceux-ci sont planifiés pour le futur et peuvent donc rejoindre le tas. Ainsi, le tas est une bonne structure pour implémenter un ordonnanceur (et c'est ce que j'ai utilisé pour mon séquenceur MIDI ☺).

Plusieurs structures ont été étudiées en détail pour implémenter des ordonnanceurs et les tas sont bien adaptés : ils sont raisonnablement rapides, leur vitesse est presque constante et le pire cas ne diffère pas trop du cas moyen. S'il existe des représentations qui sont plus efficaces en général, les pires cas peuvent être terriblement mauvais.

Les tas sont également très utiles pour ordonner les données sur de gros disques. Vous savez probablement qu'un gros tri implique la production de séquences pré-classées (dont la taille est généralement liée à la quantité de mémoire CPU disponible), suivie par une passe de fusion qui est généralement organisée de façon très intelligente¹. Il est très important que le classement initial produise des séquences les plus longues possibles. Les tournois sont une bonne façon d'arriver à ce résultat. Si, en utilisant toute la mémoire disponible pour stocker un tournoi, vous remplacez et faites percoler les éléments qui s'avèrent acceptables pour la séquence courante, vous produirez des séquences d'une taille égale au double de la mémoire pour une entrée aléatoire et bien mieux pour une entrée approximativement triée.

Qui plus est, si vous écrivez l'élément 0 sur le disque et que vous recevez en entrée un élément qui n'est pas adapté au tournoi actuel (parce que sa valeur « gagne » par rapport à la dernière valeur de sortie), alors il ne peut pas être stocké dans le tas donc la taille de ce dernier diminue. La mémoire libérée peut être réutilisée immédiatement pour progressivement construire un deuxième tas, qui croît à la même vitesse que le premier décroît. Lorsque le premier tas a complètement disparu, vous échangez les tas et démarrez une nouvelle séquence. Malin et plutôt efficace !

Pour résumer, les tas sont des structures de données qu'il est bon de connaître. Je les utilise dans quelques applications et je pense qu'il est bon de garder le module *heap* sous le coude. ☺

Notes

8.6 bisect — Algorithme de bisection de listes

Code source : [Lib/bisect.py](#)

Ce module fournit des outils pour maintenir une liste triée sans avoir à la trier à chaque insertion. Il pourrait être plus rapide que l'approche classique pour de longues listes d'éléments dont les opérations de comparaison sont lourdes. Le module se nomme *bisect* car il utilise une approche simple par bisection. Si vous voulez un exemple de cet algorithme, le mieux est d'aller lire le code source de ce module (les conditions sur les limites y étant justes !).

Les fonctions suivantes sont fournies :

`bisect.bisect_left(a, x, lo=0, hi=len(a))`

Trouve le point d'insertion de *x* dans *a* permettant de conserver l'ordre. Les paramètres *lo* et *hi* permettent de limiter les emplacements à vérifier dans la liste, par défaut toute la liste est utilisée. Si *x* est déjà présent dans *a*, le point d'insertion proposé sera avant (à gauche) de l'entrée existante. Si *a* est déjà triée, la valeur renvoyée peut directement être utilisée comme premier paramètre de `list.insert()`.

Le point d'insertion renvoyé, *i*, coupe la liste *a* en deux moitiés telles que, pour la moitié de gauche : `all(val < x for val in a[lo:i])`, et pour la partie de droite : `all(val >= x for val in a[i:hi])`.

`bisect.bisect_right(a, x, lo=0, hi=len(a))`

`bisect.bisect(a, x, lo=0, hi=len(a))`

Semblable à `bisect_left()`, mais renvoie un point d'insertion après (à droite) d'une potentielle entrée existante valant *x* dans *a*.

1. Les algorithmes de répartition de charge pour les disques, courants de nos jours, sont plus embêtants qu'utiles, en raison de la capacité des disques à réaliser des accès aléatoires. Sur les périphériques qui ne peuvent faire que de la lecture séquentielle, comme les gros lecteurs à bandes, le besoin était différent et il fallait être malin pour s'assurer (bien à l'avance) que chaque mouvement de bande serait le plus efficace possible (c'est-à-dire participerait au mieux à l'« avancée » de la fusion). Certaines cassettes pouvaient même lire à l'envers et cela était aussi utilisé pour éviter de remonter dans le temps. Croyez-moi, les bons tris sur bandes étaient spectaculaires à regarder ! Depuis la nuit des temps, trier a toujours été le Grand Art ! ☺

Le point d'insertion renvoyé, *i*, coupe la liste *a* en deux moitiés telles que, pour la moitié de gauche : `all(val <= x for val in a[lo:i])` et pour la moitié de droite : `all(val > x for val in a[i:hi])`.

`bisect.insort_left(a, x, lo=0, hi=len(a))`

Insère *x* dans *a* en conservant le tri. C'est l'équivalent de `a.insert(bisect.bisect_left(a, x, lo, hi), x)`, tant que *a* est déjà triée. Gardez en tête que bien que la recherche ne coûte que $O(\log n)$, l'insertion coûte $O(n)$.

`bisect.insort_right(a, x, lo=0, hi=len(a))`

`bisect.insort(a, x, lo=0, hi=len(a))`

Similaire à `insort_left()`, mais en insérant *x* dans *a* après une potentielle entrée existante égale à *x*.

Voir aussi :

[SortedCollection recipe](#) utilise le module `bisect` pour construire une classe collection exposant des méthodes de recherches naturelles et gérant une fonction clef. Les clefs sont pré-calculées pour économiser des appels inutiles à la fonction clef durant les recherches.

8.6.1 Chercher dans des listes triées

Les fonctions `bisect()` ci-dessus sont utiles pour insérer des éléments, mais peuvent être étranges et peu naturelles à utiliser pour rechercher des éléments. Les cinq fonctions suivantes montrent comment les transformer en recherche plus classique pour les listes triées :

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.6.2 Autres Exemples

La fonction `bisect()` peut être utile pour des recherches dans des tableaux de nombres. Cet exemple utilise `bisect()` pour rechercher la note (sous forme de lettre) correspondant à un note sous forme de points, en se basant sur une échelle prédéfinie : plus de 90 vaut “A”, de 80 à 89 vaut “B”, etc... :

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

Contrairement à la fonction `sorted()`, ça n'aurait pas de sens pour la fonction `bisect()` d'avoir un paramètre `key` ou `reversed`, qui conduiraient à une utilisation inefficace (des appels successifs à la fonction `bisect` n'auraient aucun moyen de se « souvenir » des recherches de clef précédentes).

Il est préférable d'utiliser une liste de clefs pré-calculée pour chercher l'index de l'enregistrement en question :

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]           # precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.7 array — Tableaux efficaces de valeurs numériques

Ce module définit un type d'objet qui permet de représenter de façon compacte un tableau (*array*) de valeurs élémentaires : caractères, entiers, flottants. Les tableaux sont de type séquence et se comportent de manière très similaire aux listes, sauf que les types d'objets qui y sont stockés sont limités. Le type est spécifié au moment de la création de l'objet en utilisant *type code*, qui est un caractère unique. Voir ci-dessous pour la définition des types :

Code d'indication du type	Type C	Type Python	Taille minimum en octets	Notes
'b'	signed char	<i>int</i>	1	
'B'	unsigned char	<i>int</i>	1	
'u'	Py_UNICODE	Caractère Unicode	2	(1)
'h'	signed short	<i>int</i>	2	
'H'	unsigned short	<i>int</i>	2	
'i'	signed int	<i>int</i>	2	
'I'	unsigned int	<i>int</i>	2	
'l'	signed long	<i>int</i>	4	
'L'	unsigned long	<i>int</i>	4	
'q'	signed long long	<i>int</i>	8	(2)
'Q'	unsigned long long	<i>int</i>	8	(2)
'f'	<i>float</i>	<i>float</i>	4	
'd'	double	<i>float</i>	8	

Notes :

- (1) Le code de type 'u' correspond au type obsolète de Python caractère Unicode (Py_UNICODE de type wchar_t). Selon la plateforme, il peut être 16 bits ou 32 bits.
'u' sera supprimé avec le reste de l'API Py_UNICODE.
Deprecated since version 3.3, will be removed in version 4.0.
- (2) Les codes de type 'q' et 'Q' ne sont disponibles que si le compilateur C de la plateforme utilisé pour construire Python gère le type C long long, ou, sur Windows, __int64.
Nouveau dans la version 3.3.

La représentation réelle des valeurs est déterminée par l'architecture de la machine (à proprement parler, par l'implémentation C). La taille réelle est accessible via l'attribut `itemsize`.

Le module définit le type suivant :

class `array.array` (*typecode* [, *initializer*])

Un nouveau tableau dont les éléments sont limités par *typecode*, et initialisés par la valeur optionnelle *initializer*, qui peut être une liste, un *bytes-like object*, ou un itérable sur des éléments du type approprié.

Si le paramètre *initializer* est une liste ou une chaîne de caractères, il est passé à la méthode `fromlist()`, `frombytes()` ou `fromunicode()` du tableau (voir ci-dessous) pour ajouter les éléments initiaux du tableau. Si c'est un itérable, il est passé à la méthode `extend()`.

`array.typecodes`

Une chaîne avec tous les codes de types disponibles.

Les objets de tableau supportent les opérations classiques de séquence : indiciage, découpage, concaténation et multiplication. Lors de l'utilisation de tranche, la valeur assignée doit être un tableau du même type ; dans tous les autres cas, l'exception `TypeError` est levée. Les objets de tableau implémentent également l'interface tampon, et peuvent être utilisés partout où *bytes-like objects* sont supportés.

Les éléments de données et méthodes suivants sont également supportés :

`array.typecode`

Le code (de type Python caractère) utilisé pour spécifier le type des éléments du tableau.

`array.itemsize`

La longueur en octets d'un élément du tableau dans la représentation interne.

`array.append` (*x*)

Ajoute un nouvel élément avec la valeur *x* à la fin du tableau.

`array.buffer_info()`

Renvoie un tuple (`address`, `length`) indiquant l'adresse mémoire courante et la longueur en éléments du tampon utilisé pour contenir le contenu du tableau. La taille du tampon mémoire en octets peut être calculée par `array.buffer_info()[1] * array.itemsize`. Ceci est parfois utile lorsque vous travaillez sur des interfaces E/S de bas niveau (et intrinsèquement dangereuses) qui nécessitent des adresses mémoire, telles que certaines opérations `ioctl()`. Les nombres renvoyés sont valides tant que le tableau existe et qu'aucune opération qui modifie sa taille ne lui est appliquée.

Note : Lors de l'utilisation d'objets tableaux provenant de codes écrits en C ou C++ (le seul moyen d'utiliser efficacement ces informations), il est plus logique d'utiliser l'interface tampon supportée par les objets tableaux. Cette méthode est maintenue pour des raisons de rétrocompatibilité et devrait être évitée dans un nouveau code. L'interface tampon est documentée dans `bufferobjects`.

`array.byteswap()`

Boutisme de tous les éléments du tableau. Ceci n'est pris en charge que pour les valeurs de 1, 2, 4 ou 8 octets ; pour les autres types de valeur, `RuntimeError` est levée. Il est utile lors de la lecture de données à partir d'un fichier écrit sur une machine avec un ordre d'octets différent.

`array.count(x)`

Renvoie le nombre d'occurrences de `x` dans le tableau.

`array.extend(iterable)`

Ajoute les éléments de `iterable` à la fin du tableau. Si `iterable` est un autre tableau, il doit le même code d'indication du type ; dans le cas contraire, `TypeError` sera levée. Si `iterable` n'est pas un tableau, il doit être itérable et ces éléments doivent être du bon type pour être ajoutés dans le tableau.

`array.frombytes(s)`

Ajoute des éléments de la chaîne, interprétant la chaîne comme un tableau de valeurs machine (comme si elle avait été lue depuis le fichier en utilisant la méthode `from file()`).

Nouveau dans la version 3.2 : `fromstring()` est renommée en `frombytes()` pour plus de lisibilité.

`array.fromfile(f, n)`

Lit `n` éléments (en tant que valeurs machine) du *file object* `f` et les ajouter à la fin du tableau. Si moins de `n` éléments sont disponibles, `EOFError` est levée, mais les éléments qui étaient disponibles sont tout de même insérés dans le tableau. `f` doit être un objet fichier natif ; quelque chose d'autre avec une méthode `read()` ne suffit pas.

`array.fromlist(list)`

Ajoute les éléments de la liste. C'est l'équivalent de `for x in list: a.append(x)` sauf que s'il y a une erreur de type, le tableau est inchangé.

`array.fromstring()`

Alias obsolète de `frombytes()`.

`array.fromunicode(s)`

Étend ce tableau avec les données de la chaîne Unicode donnée. Le tableau doit être de type `'u'` ; sinon `ValueError` est levée. Utiliser `array.frombytes(unicodestring.encode(enc))` pour ajouter des données Unicode à un tableau d'un autre type.

`array.index(x)`

Renvoie le plus petit `i` tel que `i` est l'index de la première occurrence de `x` dans le tableau.

`array.insert(i, x)`

Ajoute un nouvel élément avec la valeur `x` dans le tableau avant la position `i`. Les valeurs négatives sont traitées relativement à la fin du tableau.

`array.pop([i])`

Supprime l'élément du tableau avec l'index `i` et le renvoie. L'argument optionnel par défaut est à `-1`, de sorte que par défaut le dernier élément est supprimé et renvoyé.

`array.remove(x)`
 Supprime la première occurrence de *x* du tableau.

`array.reverse()`
 Inverse l'ordre des éléments du tableau.

`array.tobytes()`
 Convertit le tableau en un tableau de valeurs machine et renvoie la représentation en octets (la même séquence d'octets qui serait écrite par la méthode `tofile()`).
 Nouveau dans la version 3.2 : `tostring()` est renommé en `tobytes()` pour plus de lisibilité.

`array.tofile(f)`
 Écrit tous les éléments (en tant que valeurs machine) du *file object* *f*.

`array.tolist()`
 Convertit le tableau en une liste ordinaire avec les mêmes éléments.

`array.tostring()`
 Alias obsolète de `tobytes()`.

`array.tounicode()`
 Convertit le tableau en une chaîne Unicode. Le tableau doit être un tableau de type 'u'; sinon `ValueError` est levée. Utilisez `array.tobytes().decode(enc)` pour obtenir une chaîne Unicode depuis un tableau de tout autre type.

Quand un objet tableau est affiché ou converti en chaîne, il est représenté en tant que `array(typecode, initializer)`. *initializer* est omis si le tableau est vide, sinon c'est une chaîne si le *typecode* est 'u', sinon c'est une liste de nombres. Il est garanti que la chaîne puisse être convertie en un tableau avec le même type et la même valeur en utilisant `eval()`, tant que la classe `array` a été importée en utilisant `from array import array`. Exemples :

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

Voir aussi :

Module `struct` Empaquetage et dépaquetage de données binaires hétérogènes.

Module `xdrlib` Empaquetage et dépaquetage des données XDR (External Data Representation) telles qu'elles sont utilisées dans certains systèmes d'appels de procédures à distance (ou RPC pour *remote procedure call* en anglais).

La documentation de **Numerical Python** <<https://docs.scipy.org/doc/>> L'extension *Numeric Python* (NumPy) définit un autre type de tableau; voir <http://www.numpy.org/> pour plus d'informations sur *Numeric Python*.

8.8 weakref — Weak references

Code source : [Lib/weakref.py](#)

The `weakref` module allows the Python programmer to create *weak references* to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive : when the only remaining references to a referent are weak references, *garbage collection* is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The `WeakKeyDictionary` and `WeakValueDictionary` classes supplied by the `weakref` module are an alternative, using weak references to construct mappings that don't keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a `WeakValueDictionary`, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

`WeakKeyDictionary` and `WeakValueDictionary` use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. `WeakSet` implements the `set` interface, but keeps weak references to its elements, just like a `WeakKeyDictionary` does.

`finalize` provides a straight forward way to register a cleanup function to be called when an object is garbage collected. This is simpler to use than setting up a callback function on a raw weak reference, since the module automatically ensures that the finalizer remains alive until the object is collected.

Most programs should find that using one of these weak container types or `finalize` is all they need – it's not usually necessary to create your own weak references directly. The low-level machinery is exposed by the `weakref` module for the benefit of advanced uses.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some *file objects*, *generators*, type objects, sockets, arrays, dequeues, regular expression pattern objects, and code objects.

Modifié dans la version 3.2 : Added support for `thread.lock`, `threading.Lock`, and code objects.

Several built-in types such as `list` and `dict` do not directly support weak references but can add support through subclassing :

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)  # this object is weak referenceable
```

Other built-in types such as `tuple` and `int` do not support weak references even when subclassed (This is an implementation detail and may be different across various Python implementations.).

Extension types can easily be made to support weak references; see `weakref-support`.

class `weakref.ref(object[, callback])`

Return a weak reference to *object*. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause `None` to be returned. If *callback* is provided and not `None`, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object's `__del__()` method.

Weak references are *hashable* if the *object* is hashable. They will maintain their hash value even after the *object* was deleted. If `hash()` is called the first time only after the *object* was deleted, the call will raise `TypeError`.

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the *callback*). If either referent has been deleted, the references are equal only if the reference objects are the same object.

This is a subclassable type rather than a factory function.

`__callback__`

This read-only attribute returns the callback currently associated to the weakref. If there is no callback or if the referent of the weakref is no longer alive then this attribute will have value `None`.

Modifié dans la version 3.4 : Added the `__callback__` attribute.

`weakref.proxy(object[, callback])`

Return a proxy to *object* which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either `ProxyType` or `CallableProxyType`, depending on whether *object* is callable. Proxy objects are not *hashable* regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevent their use as dictionary keys. *callback* is the same as the parameter of the same name to the `ref()` function.

`weakref.getweakrefcount(object)`

Return the number of weak references and proxies which refer to *object*.

`weakref.getweakrefs(object)`

Return a list of all weak reference and proxy objects which refer to *object*.

`class weakref.WeakKeyDictionary([dict])`

Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

Note : Caution : Because a `WeakKeyDictionary` is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a `WeakKeyDictionary` because actions performed by the program during iteration may cause items in the dictionary to vanish « by magic » (as a side effect of garbage collection).

`WeakKeyDictionary` objects have an additional method that exposes the internal references directly. The references are not guaranteed to be « live » at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

`WeakKeyDictionary.keyrefs()`

Return an iterable of the weak references to the keys.

`class weakref.WeakValueDictionary([dict])`

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

Note : Caution : Because a `WeakValueDictionary` is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a `WeakValueDictionary` because actions performed by the program during iteration may cause items in the dictionary to vanish « by magic » (as a side effect of garbage collection).

`WeakValueDictionary` objects have an additional method that has the same issues as the `keyrefs()` method of `WeakKeyDictionary` objects.

`WeakValueDictionary.valuerefs()`

Return an iterable of the weak references to the values.

`class weakref.WeakSet([elements])`

Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

class `weakref.WeakMethod (method)`

A custom *ref* subclass which simulates a weak reference to a bound method (i.e., a method defined on a class and looked up on an instance). Since a bound method is ephemeral, a standard weak reference cannot keep hold of it. *WeakMethod* has special code to recreate the bound method until either the object or the original function dies :

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r() ()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

Nouveau dans la version 3.4.

class `weakref.finalize (obj, func, *args, **kwargs)`

Return a callable finalizer object which will be called when *obj* is garbage collected. Unlike an ordinary weak reference, a finalizer will always survive until the reference object is collected, greatly simplifying lifecycle management.

A finalizer is considered *alive* until it is called (either explicitly or at garbage collection), and after that it is *dead*. Calling a live finalizer returns the result of evaluating `func(*arg, **kwargs)`, whereas calling a dead finalizer returns *None*.

Exceptions raised by finalizer callbacks during garbage collection will be shown on the standard error output, but cannot be propagated. They are handled in the same way as exceptions raised from an object's `__del__()` method or a weak reference's callback.

When the program exits, each remaining live finalizer is called unless its *atexit* attribute has been set to false. They are called in reverse order of creation.

A finalizer will never invoke its callback during the later part of the *interpreter shutdown* when module globals are liable to have been replaced by *None*.

`__call__()`

If *self* is alive then mark it as dead and return the result of calling `func(*args, **kwargs)`. If *self* is dead then return *None*.

`detach()`

If *self* is alive then mark it as dead and return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return *None*.

`peek()`

If *self* is alive then return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return *None*.

`alive`

Property which is true if the finalizer is alive, false otherwise.

`atexit`

A writable boolean property which by default is true. When the program exits, it calls all remaining live finalizers for which *atexit* is true. They are called in reverse order of creation.

Note : It is important to ensure that *func*, *args* and *kwargs* do not own any references to *obj*, either directly or indirectly, since otherwise *obj* will never be garbage collected. In particular, *func* should not be a bound method of

obj.

Nouveau dans la version 3.4.

`weakref.ReferenceType`

The type object for weak references objects.

`weakref.ProxyType`

The type object for proxies of objects which are not callable.

`weakref.CallableProxyType`

The type object for proxies of callable objects.

`weakref.ProxyTypes`

Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

exception `weakref.ReferenceError`

Exception raised when a proxy object is used but the underlying object has been collected. This is the same as the standard `ReferenceError` exception.

Voir aussi :

PEP 205 - Weak References The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

8.8.1 Objets à références faibles

Weak reference objects have no methods and no attributes besides `ref.__callback__`. A weak reference object allows the referent to be obtained, if it still exists, by calling it :

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns `None` :

```
>>> del o, o2
>>> print(r())
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern :

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

Using a separate test for « liveness » creates race conditions in threaded applications ; another thread can cause a weak reference to become invalidated before the weak reference is called ; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of `ref` objects can be created through subclassing. This is used in the implementation of the `WeakValueDictionary` to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of `ref` can be used to store additional information about an object and affect the value that's returned when the referent is accessed :

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.8.2 Exemple

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.8.3 Finalizer Objects

The main benefit of using `finalize` is that it makes it simple to register a callback without needing to preserve the returned finalizer object. For instance

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

The finalizer can be called directly as well. However the finalizer will invoke the callback at most once.

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                                # callback not called because finalizer dead
>>> del obj                             # callback not called because finalizer dead
```

You can unregister a finalizer using its `detach()` method. This kills the finalizer and returns the arguments passed to the constructor when it was created.

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<__main__.Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

Unless you set the `atexit` attribute to `False`, a finalizer will be called when the program exits if it is still alive. For instance

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.8.4 Comparing finalizers with `__del__()` methods

Suppose we want to create a class whose instances represent temporary directories. The directories should be deleted with their contents when the first of the following events occurs :

- the object is garbage collected,
- the object's `remove()` method is called, or
- the program exits.

We might try to implement the class using a `__del__()` method as follows :

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

Starting with Python 3.4, `__del__()` methods no longer prevent reference cycles from being garbage collected, and module globals are no longer forced to `None` during *interpreter shutdown*. So this code should work without any issues on CPython.

However, handling of `__del__()` methods is notoriously implementation specific, since it depends on internal details of the interpreter's garbage collector implementation.

A more robust alternative can be to define a finalizer which only references the specific functions and objects that it needs, rather than having access to the full state of the object :

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

Defined like this, our finalizer only receives a reference to the details it needs to clean up the directory appropriately. If the object never gets garbage collected the finalizer will still be called at exit.

The other advantage of weakref based finalizers is that they can be used to register finalizers for classes where the definition is controlled by a third party, such as running code when a module is unloaded :

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
    weakref.finalize(sys.modules[__name__], unloading_module)
```

Note : If you create a finalizer object in a daemon thread just as the program exits then there is the possibility that the finalizer does not get called at exit. However, in a daemon thread `atexit.register()`, `try: ... finally: ...` and `with: ...` do not guarantee that cleanup occurs either.

8.9 types — Dynamic type creation and names for built-in types

Code source : [Lib/types.py](#)

This module defines utility function to assist in dynamic creation of new types.

It also defines names for some object types that are used by the standard Python interpreter, but not exposed as builtins like `int` or `str` are.

Finally, it provides some additional type-related utility classes and functions that are not fundamental enough to be builtins.

8.9.1 Dynamic Type Creation

`types.new_class` (*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Creates a class object dynamically using the appropriate metaclass.

The first three arguments are the components that make up a class definition header : the class name, the base classes (in order), the keyword arguments (such as `metaclass`).

The *exec_body* argument is a callback that is used to populate the freshly created class namespace. It should accept the class namespace as its sole argument and update the namespace directly with the class contents. If no callback is provided, it has the same effect as passing in `lambda ns: ns`.

Nouveau dans la version 3.3.

`types.prepare_class` (*name*, *bases=()*, *kwds=None*)

Calculates the appropriate metaclass and creates the class namespace.

The arguments are the components that make up a class definition header : the class name, the base classes (in order) and the keyword arguments (such as `metaclass`).

The return value is a 3-tuple : `metaclass, namespace, kwds`

metaclass is the appropriate metaclass, *namespace* is the prepared class namespace and *kwds* is an updated copy of the passed in *kwds* argument with any 'metaclass' entry removed. If no *kwds* argument is passed in, this will be an empty dict.

Nouveau dans la version 3.3.

Modifié dans la version 3.6 : The default value for the `namespace` element of the returned tuple has changed. Now an insertion-order-preserving mapping is used when the metaclass does not have a `__prepare__` method,

Voir aussi :

metaclasses Full details of the class creation process supported by these functions

PEP 3115 — Méta-classes dans Python 3000 introduction de la fonction automatique `__prepare__` de l'espace de nommage

8.9.2 Standard Interpreter Types

This module provides names for many of the types that are required to implement a Python interpreter. It deliberately avoids including some of the types that arise only incidentally during processing such as the `listiterator` type.

Typical use of these names is for `isinstance()` or `issubclass()` checks.

Standard names are defined for the following types :

`types.FunctionType`

`types.LambdaType`

The type of user-defined functions and functions created by `lambda` expressions.

`types.GeneratorType`

The type of *generator*-iterator objects, created by generator functions.

`types.CoroutineType`

The type of *coroutine* objects, created by `async def` functions.

Nouveau dans la version 3.5.

`types.AsyncGeneratorType`

The type of *asynchronous generator*-iterator objects, created by asynchronous generator functions.

Nouveau dans la version 3.6.

`types.CodeType`

The type for code objects such as returned by `compile()`.

`types.MethodType`

The type of methods of user-defined class instances.

`types.BuiltinFunctionType`

`types.BuiltinMethodType`

The type of built-in functions like `len()` or `sys.exit()`, and methods of built-in classes. (Here, the term « built-in » means « written in C ».)

class `types.ModuleType` (*name*, *doc=None*)

The type of *modules*. Constructor takes the name of the module to be created and optionally its *docstring*.

Note : Use `importlib.util.module_from_spec()` to create a new module if you wish to set the various import-controlled attributes.

`__doc__`

The *docstring* of the module. Defaults to `None`.

`__loader__`

The *loader* which loaded the module. Defaults to `None`.

Modifié dans la version 3.4 : Defaults to `None`. Previously the attribute was optional.

`__name__`

The name of the module.

`__package__`

Which *package* a module belongs to. If the module is top-level (i.e. not a part of any specific package) then the attribute should be set to `' '`, else it should be set to the name of the package (which can be `__name__` if the module is a package itself). Defaults to `None`.

Modifié dans la version 3.4 : Defaults to `None`. Previously the attribute was optional.

`types.TracebackType`

The type of traceback objects such as found in `sys.exc_info()[2]`.

`types.FrameType`

The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

types.GetSetDescriptorType

The type of objects defined in extension modules with `PyGetSetDef`, such as `FrameType.f_locals` or `array.array.typecode`. This type is used as descriptor for object attributes; it has the same purpose as the `property` type, but for classes defined in extension modules.

types.MemberDescriptorType

The type of objects defined in extension modules with `PyMemberDef`, such as `datetime.timedelta.days`. This type is used as descriptor for simple C data members which use standard conversion functions; it has the same purpose as the `property` type, but for classes defined in extension modules.

CPython implementation detail : In other implementations of Python, this type may be identical to `GetSetDescriptorType`.

class types.MappingProxyType(mapping)

Read-only proxy of a mapping. It provides a dynamic view on the mapping's entries, which means that when the mapping changes, the view reflects these changes.

Nouveau dans la version 3.3.

key in proxy

Return True if the underlying mapping has a key *key*, else False.

proxy[key]

Return the item of the underlying mapping with key *key*. Raises a `KeyError` if *key* is not in the underlying mapping.

iter(proxy)

Return an iterator over the keys of the underlying mapping. This is a shortcut for `iter(proxy.keys())`.

len(proxy)

Return the number of items in the underlying mapping.

copy()

Return a shallow copy of the underlying mapping.

get(key[, default])

Return the value for *key* if *key* is in the underlying mapping, else *default*. If *default* is not given, it defaults to None, so that this method never raises a `KeyError`.

items()

Return a new view of the underlying mapping's items ((key, value) pairs).

keys()

Return a new view of the underlying mapping's keys.

values()

Return a new view of the underlying mapping's values.

8.9.3 Additional Utility Classes and Functions

class types.SimpleNamespace

A simple `object` subclass that provides attribute access to its namespace, as well as a meaningful repr.

Unlike `object`, with `SimpleNamespace` you can add and remove attributes. If a `SimpleNamespace` object is initialized with keyword arguments, those are directly added to the underlying namespace.

The type is roughly equivalent to the following code :

```
class SimpleNamespace:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}".format(k, self.__dict__[k]) for k in keys)
```

(suite sur la page suivante)

(suite de la page précédente)

```
return "{}({})".format(type(self).__name__, ", ".join(items))

def __eq__(self, other):
    return self.__dict__ == other.__dict__
```

`SimpleNamespace` may be useful as a replacement for `class NS: pass`. However, for a structured record type use `namedtuple()` instead.

Nouveau dans la version 3.3.

`types.DynamicClassAttribute` (*fget=None, fset=None, fdel=None, doc=None*)

Route attribute access on a class to `__getattr__`.

This is a descriptor, used to define attributes that act differently when accessed through an instance and through a class. Instance access remains normal, but access to an attribute through a class will be routed to the class's `__getattr__` method; this is done by raising `AttributeError`.

This allows one to have properties active on an instance, and have virtual attributes on the class with the same name (see `Enum` for an example).

Nouveau dans la version 3.4.

8.9.4 Coroutine Utility Functions

`types.coroutine` (*gen_func*)

This function transforms a *generator* function into a *coroutine function* which returns a generator-based coroutine. The generator-based coroutine is still a *generator iterator*, but is also considered to be a *coroutine* object and is *awaitable*. However, it may not necessarily implement the `__await__()` method.

If *gen_func* is a generator function, it will be modified in-place.

If *gen_func* is not a generator function, it will be wrapped. If it returns an instance of `collections.abc.Generator`, the instance will be wrapped in an *awaitable* proxy object. All other types of objects will be returned as is.

Nouveau dans la version 3.5.

8.10 copy — Opérations de copie superficielle et récursive

Code source : [Lib/copy.py](#)

Les instructions d'affectation en Python ne copient pas les objets, elles créent des liens entre la cible et l'objet. Concernant les collections qui sont muables ou contiennent des éléments muables, une copie est parfois nécessaire, pour pouvoir modifier une copie sans modifier l'autre. Ce module met à disposition des opérations de copie génériques superficielle et récursive (comme expliqué ci-dessous).

Résumé de l'interface :

`copy.copy` (*x*)

Renvoie une copie superficielle de *x*.

`copy.deepcopy` (*x*, [*memo*])

Renvoie une copie récursive de *x*.

exception `copy.error`

Levée pour les erreurs spécifiques au module.

La différence entre copie superficielle et récursive n'est pertinente que pour les objets composés (objets contenant d'autres objets, comme des listes ou des instances de classe) :

- Une *copie superficielle* construit un nouvel objet composé puis (dans la mesure du possible) insère dans l'objet composé des *références* aux objets trouvés dans l'original.
- Une *copie récursive (ou profonde)* construit un nouvel objet composé puis, récursivement, insère dans l'objet composé des *copies* des objets trouvés dans l'objet original.

On rencontre souvent deux problèmes avec les opérations de copie récursive qui n'existent pas avec les opérations de copie superficielle :

- Les objets récursifs (objets composés qui, directement ou indirectement, contiennent une référence à eux-mêmes) peuvent causer une boucle récursive.
- Comme une copie récursive copie tout, elle peut en copier trop, par exemple des données qui sont destinées à être partagées entre différentes copies.

La fonction `deepcopy()` évite ces problèmes en :

- keeping a memo dictionary of objects already copied during the current copying pass ; and
- laissant les classes créées par l'utilisateur écraser l'opération de copie ou l'ensemble de composants copiés.

Ce module ne copie pas les types tels que module, méthode, trace d'appels, cadre de pile, fichier, socket, fenêtre, tableau, ou tout autre type similaire. Il « copie » les fonctions et les classes (superficiellement et récursivement), en retournant l'objet original inchangé ; c'est compatible avec la manière dont ils sont traités par le module `pickle`.

Les copies superficielles de dictionnaires peuvent être faites en utilisant `dict.copy()`, et de listes en affectant un slice de la liste, par exemple, `copied_list = original_list[:]`.

Les classes peuvent utiliser les mêmes interfaces de contrôle que celles utilisées pour la sérialisation. Voir la description du module `pickle` pour plus d'informations sur ces méthodes. En effet, le module `copy` utilise les fonctions de sérialisation enregistrées à partir du module `copyreg`.

In order for a class to define its own copy implementation, it can define special methods `__copy__()` and `__deepcopy__()`. The former is called to implement the shallow copy operation ; no additional arguments are passed. The latter is called to implement the deep copy operation ; it is passed one argument, the memo dictionary. If the `__deepcopy__()` implementation needs to make a deep copy of a component, it should call the `deepcopy()` function with the component as first argument and the memo dictionary as second argument.

Voir aussi :

Module `pickle` Discussion sur les méthodes spéciales utilisées pour gérer la récupération et la restauration de l'état d'un objet.

8.11 pprint — L'affichage élégant de données

Code source : `Lib/pprint.py`

Le module `pprint` permet « d'afficher élégamment » des structures de données Python arbitraires sous une forme qui peut être utilisée ensuite comme une entrée dans l'interpréteur. Si les structures formatées incluent des objets qui ne sont pas des types Python fondamentaux, leurs représentations peuvent ne pas être acceptables en tant que telles par l'interpréteur. Cela peut être le cas si des objets tels que des fichiers, des interfaces de connexion (*sockets* en anglais) ou des classes sont inclus, c'est aussi valable pour beaucoup d'autres types d'objets qui ne peuvent être représentés sous forme littérale en Python.

L'affichage formaté affiche tant que possible les objets sur une seule ligne, et les sépare sur plusieurs lignes s'ils dépassent la largeur autorisée par l'interpréteur. Créez explicitement des objets `PrettyPrinter` si vous avez besoin de modifier les limites de largeur.

Les dictionnaires sont classés par clés avant que l'affichage ne soit calculé.

Le module `pprint` définit une seule classe :

class pprint.PrettyPrinter(indent=1, width=80, depth=None, stream=None, *, compact=False)

Crée une instance de `PrettyPrinter`. Ce constructeur accepte plusieurs paramètres nommés. Un flux de sortie peut être défini par le mot clé `stream`; la seule méthode utilisée sur l'objet `stream` est la méthode `write()` du protocole de fichiers. Si rien n'est spécifié, la classe `PrettyPrinter` utilise `sys.stdout`. La taille de l'indentation ajoutée à chaque niveau récursif est spécifiée par `indent`; la valeur par défaut vaut 1. D'autres valeurs pourraient donner des résultats surprenants, mais peuvent aider à mieux visualiser les imbrications. Le nombre de niveaux qui peuvent être affichés est contrôlé par `*depth*`; si la structure de données est trop profonde, le niveau suivant est remplacé par ```....`. Par défaut il n'y a pas de contraintes sur la profondeur des objets formatés. Vous pouvez limiter la largeur de la sortie à l'aide du paramètre `width`; la valeur par défaut est de 80 caractères. Si une structure ne peut pas être formatée dans les limites de la largeur contrainte, le module fait au mieux. Si `compact` est initialisé à `False` (la valeur par défaut), chaque élément d'une longue séquence est formaté sur une ligne séparée. Si `compact` est initialisé à `True`, tous les éléments qui peuvent tenir dans la largeur définie sont formatés sur chaque ligne de sortie.

Modifié dans la version 3.4 : Ajout du paramètre `compact`.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[
    ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

Le module `pprint` fournit aussi quelques fonctions de raccourcis :

pprint.pformat (object, indent=1, width=80, depth=None, *, compact=False)

Renvoie une représentation formatée de `object` sous forme de chaîne de caractères. `indent`, `width`, `depth` et `compact` sont passés au constructeur de `PrettyPrinter` comme paramètres de formatage.

Modifié dans la version 3.4 : Ajout du paramètre `compact`.

pprint.pprint (object, stream=None, indent=1, width=80, depth=None, *, compact=False)

Affiche la représentation formatée de `object` sur `stream`, suivie d'un retour à la ligne. Si `stream` vaut `None`, `sys.stdout` est alors utilisé. Vous pouvez l'utiliser dans l'interpréteur interactif de Python au lieu de la fonction `print()` pour inspecter les valeurs (vous pouvez même réassigner `print = pprint.pprint` pour une utilisation au sein de sa portée). `indent`, `width`, `depth` et `compact` sont passés au constructeur de classe `PrettyPrinter` comme paramètres de formatage.

Modifié dans la version 3.4 : Ajout du paramètre `compact`.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

`pprint.isreadable(object)`

Détermine si la représentation formatée de *object* est « lisible », ou s'il peut être utilisé pour recomposer sa valeur en utilisant la fonction `eval()`. Cela renvoie toujours `False` pour les objets récurifs.

```
>>> pprint.isreadable(stuff)
False
```

`pprint.isrecursive(object)`

Détermine si *object* requiert une représentation récursive.

Une dernière fonction de support est définie ainsi :

`pprint.saferepr(object)`

Renvoie une représentation de *object* sous forme de chaîne de caractère, celle-ci est protégée contre les structures de données récurives. Si la représentation de *object* présente une entrée récursive, celle-ci sera représentée telle que `<Recursion on typename with id=number>`. Par ailleurs, la représentation de l'objet n'est pas formatée.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.11.1 Les Objets PrettyPrinter

Les instances de la classe `PrettyPrinter` ont les méthodes suivantes :

`PrettyPrinter.pformat(object)`

Renvoie la représentation formatée de *object*. Cela prend en compte les options passées au constructeur de la classe `PrettyPrinter`.

`PrettyPrinter.pprint(object)`

Affiche sur le flux configuré la représentation formatée de *object*, suivie d'une fin de ligne.

Les méthodes suivantes fournissent les implémentations pour les fonctions correspondantes de mêmes noms. L'utilisation de ces méthodes sur une instance est légèrement plus efficace, car les nouveaux objets `PrettyPrinter` n'ont pas besoin d'être créés.

`PrettyPrinter.isreadable(object)`

Détermine si la représentation formatée de *object* est « lisible », ou si elle peut être utilisée pour recomposer sa valeur en utilisant la fonction `eval()`. Cela renvoie toujours `False` pour les objets récurifs. Si le paramètre *depth* de la classe `PrettyPrinter` est initialisé et que l'objet est plus « profond » que permis, cela renvoie `False`.

`PrettyPrinter.isrecursive(object)`

Détermine si l'objet nécessite une représentation récursive.

Cette méthode est fournie sous forme de point d'entrée ou méthode (à déclenchement) automatique (*hook* en anglais) pour permettre aux sous-classes de modifier la façon dont les objets sont convertis en chaînes. L'implémentation par défaut est celle de la fonction `saferepr()`.

`PrettyPrinter.format(object, context, maxlevels, level)`

Renvoie trois valeurs : la version formatée de *object* sous forme de chaîne de caractères, une option indiquant si le résultat est « lisible », et une option indiquant si une récursion a été détectée. Le premier argument est l'objet à représenter. Le deuxième est un dictionnaire qui contient l'"*id()*" des objets (conteneurs directs ou indirects de *objet* qui affectent sa représentation) qui font partie du contexte de représentation courant tel que les clés ; si un objet doit être représenté, mais l'a déjà été dans ce contexte, le troisième argument renvoie `True`. Des appels récursifs à la méthode `format()` doivent ajouter des entrées additionnelles aux conteneurs de ce dictionnaire. Le troisième argument *maxlevels*, donne la limite maximale de récursivité ; la valeur par défaut est 0. Cet argument doit être passé non modifié pour des appels non récursifs. Le quatrième argument, *level*, donne le niveau de récursivité courant ; les appels récursifs doivent être passés à une valeur inférieure à celle de l'appel courant.

8.11.2 Exemple

Pour illustrer quelques cas pratiques de l'utilisation de la fonction `pprint()` et de ses paramètres, allons chercher des informations sur un projet `PyPI` :

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/json') as resp:
...     project_info = json.load(resp)['info']
```

Dans sa forme basique, la fonction `pprint()` affiche l'intégralité de l'objet :

```
>>> pprint.pprint(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                  'Intended Audience :: Developers',
                  'License :: OSI Approved :: MIT License',
                  'Programming Language :: Python :: 2',
                  'Programming Language :: Python :: 2.6',
                  'Programming Language :: Python :: 2.7',
                  'Programming Language :: Python :: 3',
                  'Programming Language :: Python :: 3.2',
                  'Programming Language :: Python :: 3.3',
                  'Programming Language :: Python :: 3.4',
                  'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=====\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'}
```

(suite sur la page suivante)

(suite de la page précédente)

```

        'may be appropriate.',
'description_content_type': None,
'docs_url': None,
'download_url': 'UNKNOWN',
'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
'home_page': 'https://github.com/pypa/sampleproject',
'keywords': 'sample setuptools development',
'license': 'MIT',
'maintainer': None,
'maintainer_email': None,
'name': 'sampleproject',
'package_url': 'https://pypi.org/project/sampleproject/',
'platform': 'UNKNOWN',
'project_url': 'https://pypi.org/project/sampleproject/',
'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
'requires_dist': None,
'requires_python': None,
'summary': 'A sample Python project',
'version': '1.2.0'}

```

Le résultat peut être limité à une certaine profondeur en initialisant *depth*. (... est utilisé pour des contenus plus « profonds ») :

```

>>> pprint.pprint(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
               '=====\n'
               '\n'
               'This is the description file for the project.\n'
               '\n'
               'The file should use UTF-8 encoding and be written using '
               'ReStructured Text. It\n'
               'will be used to generate the project webpage on PyPI, and '
               'should be written for\n'
               'that purpose.\n'
               '\n'
               'Typical contents for this file would include an overview of '
               'the project, basic\n'
               'usage examples, etc. Generally, including the project '
               'changelog in here is not\n'
               'a good idea, although a simple "What\'s New" section for the '
               'most recent version\n'
               'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,

```

(suite sur la page suivante)

(suite de la page précédente)

```
'maintainer_email': None,
'name': 'sampleproject',
'package_url': 'https://pypi.org/project/sampleproject/',
'platform': 'UNKNOWN',
'project_url': 'https://pypi.org/project/sampleproject/',
'project_urls': {...},
'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
'requires_dist': None,
'requires_python': None,
'summary': 'A sample Python project',
'version': '1.2.0'}
```

De plus, une valeur maximale de caractères sur une ligne peut être définie en initialisant le paramètre *width*. Si un long objet ne peut être scindé, la valeur donnée à *width* sera outrepassée :

```
>>> pprint.pprint(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
               '=====\n'
               '\n'
               'This is the description file for the '
               'project.\n'
               '\n'
               'The file should use UTF-8 encoding and be '
               'written using ReStructured Text. It\n'
               'will be used to generate the project '
               'webpage on PyPI, and should be written '
               'for\n'
               'that purpose.\n'
               '\n'
               'Typical contents for this file would '
               'include an overview of the project, '
               'basic\n'
               'usage examples, etc. Generally, including '
               'the project changelog in here is not\n'
               'a good idea, although a simple "What\'s '
               'New" section for the most recent version\n'
               'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/'}
```

(suite sur la page suivante)

(suite de la page précédente)

```
'requires_dist': None,
'requires_python': None,
'summary': 'A sample Python project',
'version': '1.2.0'}
```

8.12 reprlib — Alternate repr() implementation

Code source : [Lib/reprlib.py](#)

The `reprlib` module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function :

class `reprlib.Repr`

Class which provides formatting services useful in implementing functions similar to the built-in `repr()` ; size limits for different object types are added to avoid the generation of representations which are excessively long.

`reprlib.aRepr`

This is an instance of `Repr` which is used to provide the `repr()` function described below. Changing the attributes of this object will affect the size limits used by `repr()` and the Python debugger.

`reprlib.repr(obj)`

This is the `repr()` method of `aRepr`. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.

In addition to size-limiting tools, the module also provides a decorator for detecting recursive calls to `__repr__()` and substituting a placeholder string instead.

`@reprlib.recursive_repr(fillvalue="...")`

Decorator for `__repr__()` methods to detect recursive calls within the same thread. If a recursive call is made, the `fillvalue` is returned, otherwise, the usual `__repr__()` call is made. For example :

```
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

Nouveau dans la version 3.2.

8.12.1 Repr Objects

Repr instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

`Repr.maxlevel`

Depth limit on the creation of recursive representations. The default is 6.

`Repr.maxdict`

`Repr.maxlist`

`Repr.maxtuple`

`Repr.maxset`

`Repr.maxfrozenset`

`Repr.maxdeque`

`Repr.maxarray`

Limits on the number of entries represented for the named object type. The default is 4 for *maxdict*, 5 for *maxarray*, and 6 for the others.

`Repr.maxlong`

Maximum number of characters in the representation for an integer. Digits are dropped from the middle. The default is 40.

`Repr.maxstring`

Limit on the number of characters in the representation of the string. Note that the « normal » representation of the string is used as the character source : if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is 30.

`Repr.maxother`

This limit is used to control the size of object types for which no specific formatting method is available on the *Repr* object. It is applied in a similar manner as *maxstring*. The default is 20.

`Repr.repr(obj)`

The equivalent to the built-in *repr()* that uses the formatting imposed by the instance.

`Repr.repr1(obj, level)`

Recursive implementation used by *repr()*. This uses the type of *obj* to determine which formatting method to call, passing it *obj* and *level*. The type-specific methods should call *repr1()* to perform recursive formatting, with *level - 1* for the value of *level* in the recursive call.

`Repr.repr_TYPE(obj, level)`

Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, **TYPE** is replaced by `'_'.join(type(obj).__name__.split())`. Dispatch to these methods is handled by *repr1()*. Type-specific methods which need to recursively format a value should call `self.repr1(subobj, level - 1)`.

8.12.2 Subclassing Repr Objects

The use of dynamic dispatching by *Repr.repr1()* allows subclasses of *Repr* to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added :

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
```

(suite sur la page suivante)

(suite de la page précédente)

```

    if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
        return obj.name
    return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))           # prints '<stdin>'

```

8.13 enum — Énumérations

Nouveau dans la version 3.4.

Code source : [Lib/enum.py](#)

Une énumération est un ensemble de noms symboliques, appelés *membres*, liés à des valeurs constantes et uniques. Au sein d'une énumération, les membres peuvent être comparés entre eux et il est possible d'itérer sur l'énumération elle-même.

8.13.1 Contenu du module

Ce module définit quatre classes d'énumération qui permettent de définir des ensembles uniques de noms et de valeurs : *Enum*, *IntEnum*, *Flag* et *IntFlag*. Il fournit également un décorateur, *unique()*, ainsi qu'une classe utilitaire, *auto*.

class enum.**Enum**

Classe de base pour créer une énumération de constantes. Voir la section *API par fonction* pour une syntaxe alternative de construction.

class enum.**IntEnum**

Classe de base pour créer une énumération de constantes qui sont également des sous-classes de *int*.

class enum.**IntFlag**

Classe de base pour créer une énumération de constantes pouvant être combinées avec des opérateurs de comparaison bit-à-bit, sans perdre leur qualité de *IntFlag*. Les membres de *IntFlag* sont aussi des sous-classes de *int*.

class enum.**Flag**

Classe de base pour créer une énumération de constantes pouvant être combinées avec des opérateurs de comparaison bit-à-bit, sans perdre leur qualité de *Flag*.

enum.**unique()**

Décorateur de classe qui garantit qu'une valeur ne puisse être associée qu'à un seul nom.

class enum.**auto**

Les instances de cette classe remplacent les membres d'une *Enum* par une valeur appropriée.

Nouveau dans la version 3.6 : *Flag*, *IntFlag*, *auto*

8.13.2 Création d'une *Enum*

Une énumération est créée comme une `class`, ce qui la rend facile à lire et à écrire. Une autre méthode de création est décrite dans *API par fonction*. Pour définir une énumération, il faut hériter de *Enum* de la manière suivante :

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
... 
```

Note : Valeurs des membres d'une *Enum*

La valeur d'un membre peut être de n'importe quel type : *int*, *str*, etc. Si la valeur exacte n'a pas d'importance, utilisez des instances de *auto* et une valeur appropriée sera choisie pour vous. Soyez vigilant si vous mélangez *auto* avec d'autres valeurs.

Note : Nomenclature

- La classe `Color` est une *énumération* (ou un *enum*).
 - Les attributs `Color.RED`, `Color.GREEN`, etc., sont les *membres de l'énumération* (ou les *membres de l'enum*) et sont fonctionnellement des constantes.
 - Les membres de l'*enum* ont chacun un *nom* et une *valeur* ; le nom de `Color.RED` est `RED`, la valeur de `Color.BLUE` est `3`, etc.
-

Note : Même si on utilise la syntaxe en `class` pour créer des énumérations, les *Enums* ne sont pas des vraies classes Python. Voir *En quoi les Enums sont différentes ?* pour plus de détails.

Les membres d'une énumération ont une représentation en chaîne de caractères compréhensible par un humain :

```
>>> print(Color.RED)
Color.RED
```

... tandis que leur `repr` contient plus d'informations :

```
>>> print(repr(Color.RED))
<Color.RED: 1>
```

Le *type* d'un membre est l'énumération auquel ce membre appartient :

```
>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
True
>>>
```

Les membres ont également un attribut qui contient leur nom :

```
>>> print(Color.RED.name)
RED
```

Les énumérations sont itérables, l'ordre d'itération est celui dans lequel les membres sont déclarés :

```
>>> class Shake(Enum):
...     VANILLA = 7
...     CHOCOLATE = 4
...     COOKIES = 9
...     MINT = 3
...
>>> for shake in Shake:
...     print(shake)
...
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
```

Les membres d'une énumération sont hachables, ils peuvent ainsi être utilisés dans des dictionnaires ou des ensembles :

```
>>> apples = {}
>>> apples[Color.RED] = 'red delicious'
>>> apples[Color.GREEN] = 'granny smith'
>>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
True
```

8.13.3 Accès dynamique aux membres et à leurs attributs

Il est parfois utile de pouvoir accéder dynamiquement aux membres d'une énumération (p. ex. dans des situations où il ne suffit pas d'utiliser `Color.RED` car la couleur précise n'est pas connue à l'écriture du programme). `Enum` permet de tels accès :

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

Pour accéder aux membres par leur *nom*, utilisez l'accès par indexation :

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

Pour obtenir l'attribut `name` ou `value` d'un membre :

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

8.13.4 Duplication de membres et de valeurs

Il n'est pas possible d'avoir deux membres du même nom dans un *enum* :

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

Cependant deux membres peuvent avoir la même valeur. Si deux membres A et B ont la même valeur (et que A est défini en premier), B sera un alias de A. Un accès par valeur avec la valeur commune à A et B renverra A. Un accès à B par nom renverra aussi A :

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

Note : Il est interdit de créer un membre avec le même nom qu'un attribut déjà défini (un autre membre, une méthode, etc.) ou de créer un attribut avec le nom d'un membre.

8.13.5 Coercition d'unicité des valeurs d'une énumération

Par défaut, les énumérations autorisent les alias de nom pour une même valeur. Quand ce comportement n'est pas désiré, il faut utiliser le décorateur suivant pour s'assurer que chaque valeur n'est utilisée qu'une seule fois au sein de l'énumération :

`@enum.unique`

Un décorateur de `class` spécifique aux énumérations. Il examine l'attribut `__members__` d'une énumération et recherche des alias ; s'il en trouve, l'exception `ValueError` est levée avec des détails :

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

8.13.6 Valeurs automatiques

Si la valeur exacte n'a pas d'importance, vous pouvez utiliser `auto` :

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

Les valeurs sont déterminées par `_generate_next_value_()`, qui peut être redéfinie :

```
>>> class AutoName(Enum):
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAST'>, <Ordinal.
↪WEST: 'WEST'>]
```

Note : La méthode par défaut `_generate_next_value_()` doit fournir le `int` suivant de la séquence en fonction du dernier `int` fourni, mais la séquence générée dépend de l'implémentation Python.

8.13.7 Itération

Itérer sur les membres d'une énumération ne parcourt pas les alias :

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

L'attribut spécial `__members__` est un dictionnaire ordonné qui fait correspondre les noms aux membres. Il inclut tous les noms définis dans l'énumération, alias compris :

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

L'attribut `__members__` peut servir à accéder dynamiquement aux membres de l'énumération. Par exemple, pour trouver tous les alias :

```
>>> [name for name, member in Shape.__members__.items() if member.name != name]
['ALIAS_FOR_SQUARE']
```

8.13.8 Comparaisons

Les membres d'une énumération sont comparés par identité :

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

Les comparaisons d'ordre entre les valeurs d'une énumération n'existent *pas*; les membres d'un *enum* ne sont pas des entiers (voir cependant *IntEnum* ci-dessous) :

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and 'Color'
```

A contrario, les comparaisons d'égalité existent :

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

Les comparaisons avec des valeurs ne provenant pas d'énumérations sont toujours fausses (ici encore, *IntEnum* a été conçue pour fonctionner différemment, voir ci-dessous) :

```
>>> Color.BLUE == 2
False
```

8.13.9 Membres et attributs autorisés dans une énumération

Les exemples précédents utilisent des entiers pour énumérer les valeurs. C'est un choix concis et pratique (et implémenté par défaut dans l'*API par fonction*), mais ce n'est pas une obligation. Dans la majorité des cas, il importe peu de connaître la valeur réelle d'une énumération. Il est toutefois possible de donner une valeur arbitraire aux énumérations, si cette valeur est *vraiment* significative.

Les énumérations sont des classes Python et peuvent donc avoir des méthodes et des méthodes spéciales. L'énumération suivante :

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # self is the member here
```

(suite sur la page suivante)

(suite de la page précédente)

```

...     return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
...

```

amène :

```

>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
>>> Mood.HAPPY.describe()
('HAPPY', 3)
>>> str(Mood.FUNKY)
'my custom str! 1'

```

The rules for what is allowed are as follows : names that start and end with a single underscore are reserved by enum and cannot be used ; all other attributes defined within an enumeration will become members of this enumeration, with the exception of special methods (`__str__()`, `__add__()`, etc.) and descriptors (methods are also descriptors).

Remarque : si l'énumération définit `__new__()` ou `__init__()`, alors la (ou les) valeur affectée au membre sera passée à ces méthodes. Voir l'exemple de *Planet*.

8.13.10 Restricted subclassing of enumerations

Subclassing an enumeration is allowed only if the enumeration does not define any members. So this is forbidden :

```

>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations

```

Mais celui-ci est correct :

```

>>> class Foo(Enum):
...     def some_behavior(self):
...         pass
...
>>> class Bar(Foo):
...     HAPPY = 1
...     SAD = 2
...

```

Autoriser l'héritage d'*enums* définissant des membres violerait des invariants sur les types et les instances. D'un autre côté, il est logique d'autoriser un groupe d'énumérations à partager un comportement commun (voir par exemple *OrderedEnum*).

8.13.11 Sérialisation

Les énumérations peuvent être sérialisées et désérialisées :

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

Les restrictions habituelles de sérialisation s'appliquent : les *enums* à sérialiser doivent être déclarés dans l'espace de nom de haut niveau du module car la désérialisation nécessite que ces *enums* puissent être importés depuis ce module.

Note : Depuis la version 4 du protocole de *pickle*, il est possible de sérialiser facilement des *enums* imbriqués dans d'autres classes.

Redéfinir la méthode `__reduce_ex__()` permet de modifier la sérialisation ou la dé-sérialisation des membres d'une énumération.

8.13.12 API par fonction

La *Enum* est callable et implémente l'API par fonction suivante :

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>, <Animal.DOG: 4>]
```

La sémantique de cette API est similaire à *namedtuple*. Le premier argument de l'appel à *Enum* est le nom de l'énumération.

Le second argument est la *source* des noms des membres de l'énumération. Il peut être une chaîne de caractères contenant les noms séparés par des espaces, une séquence de noms, une séquence de couples clé / valeur ou un dictionnaire (p. ex. un *dict*) de valeurs indexées par des noms. Les deux dernières options permettent d'affecter des valeurs arbitraires aux énumérations ; les autres affectent automatiquement des entiers en commençant par 1 (le paramètre *start* permet de changer la valeur de départ). Ceci renvoie une nouvelle classe dérivée de *Enum*. En d'autres termes, la déclaration de *Animal* ci-dessus équivaut à :

```
>>> class Animal(Enum) :
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
... 
```

La valeur de départ par défaut est 1 et non 0 car 0 au sens booléen vaut *False* alors que tous les membres d'une *enum* valent *True*.

La sérialisation d'énumérations créées avec l'API en fonction peut être source de problèmes, car celle-ci repose sur des détails d'implémentation de l'affichage de la pile d'appel pour tenter de déterminer dans quel module l'énumération est

créée (p. ex. elle échouera avec les fonctions utilitaires provenant d'un module séparé et peut ne pas fonctionner avec IronPython ou Jython). La solution consiste à préciser explicitement le nom du module comme ceci :

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

Avertissement : Si `module` n'est pas fourni et que `Enum` ne peut pas le deviner, les nouveaux membres de `l'Enum` ne seront pas désérialisables ; pour garder les erreurs au plus près de leur origine, la sérialisation sera désactivée.

Le nouveau protocole version 4 de `pickle` se base lui aussi, dans certains cas, sur le fait que `__qualname__` pointe sur l'endroit où `pickle` peut trouver la classe. Par exemple, si la classe était disponible depuis la classe `SomeData` dans l'espace de nom de plus haut niveau :

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='SomeData.Animal')
```

La signature complète est la suivante :

```
Enum(value='NewEnumName', names=<...>, *, module='...', qualname='...', type=<mixed-
↳ in class>, start=1)
```

value Le nom de la nouvelle classe `Enum`.

names Les membres de l'énumération. Une chaîne de caractères séparés par des espaces ou des virgules (la valeur de départ est fixée à 1, sauf si spécifiée autrement) :

```
'RED GREEN BLUE' | 'RED, GREEN, BLUE' | 'RED, GREEN, BLUE'
```

ou un itérateur sur les noms :

```
['RED', 'GREEN', 'BLUE']
```

ou un itérateur sur les tuples (nom, valeur) :

```
[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]
```

ou une correspondance :

```
{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}
```

module nom du module dans lequel la classe `Enum` se trouve.

qualname localisation de la nouvelle classe `Enum` dans le module.

type le type à mélanger dans la nouvelle classe `Enum`.

start index de départ si uniquement des noms sont passés.

Modifié dans la version 3.5 : Ajout du paramètre `start`.

8.13.13 Énumérations dérivées

IntEnum

La première version dérivée de `Enum` qui existe est aussi une sous-classe de `int`. Les membres de `IntEnum` peuvent être comparés à des entiers et, par extension, les comparaisons entre des énumérations entières de type différent sont possibles :

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

Elles ne peuvent cependant toujours pas être comparées à des énumérations standards de *Enum* :

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

Les valeurs de *IntEnum* se comportent comme des entiers, comme on pouvait s'y attendre :

```
>>> int(Shape.CIRCLE)
1
>>> ['a', 'b', 'c'][Shape.CIRCLE]
'b'
>>> [i for i in range(Shape.SQUARE)]
[0, 1]
```

IntFlag

La version dérivée suivante de *Enum* est *IntFlag*. Elle est aussi basée sur *int*, à la différence près que les membres de *IntFlag* peuvent être combinés en utilisant les opérateurs bit-à-bit (&, |, ^, ~) et que le résultat reste un membre de *IntFlag*. Cependant, comme le nom l'indique, les membres d'une classe *IntFlag* héritent aussi de *int* et peuvent être utilisés là où un *int* est utilisé. Toute opération sur un membre d'une classe *IntFlag*, autre qu'un opérateur bit-à-bit lui fait perdre sa qualité de *IntFlag*.

Nouveau dans la version 3.6.

Exemple d'une classe *IntFlag* :

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

Il est aussi possible de nommer les combinaisons :

```
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm.-8: -8>
```

Une autre différence importante entre *IntFlag* et *Enum* est que, si aucune option n'est activée (la valeur vaut 0), son évaluation booléenne est *False* :

```
>>> Perm.R & Perm.X
<Perm.0: 0>
>>> bool(Perm.R & Perm.X)
False
```

Comme les membres d'une classe *IntFlag* héritent aussi de *int*, ils peuvent être combinés avec eux :

```
>>> Perm.X | 8
<Perm.8|X: 9>
```

Option

La dernière version dérivée est la classe *Flag*. Comme *IntFlag*, les membres d'une classe *Flag* peuvent être combinés en utilisant les opérateurs de comparaison bit-à-bit. Cependant, à la différence de *IntFlag*, ils ne peuvent ni être combinés, ni être comparés avec une autre énumération *Flag*, ni avec *int*. Bien qu'il soit possible de définir directement les valeurs, il est recommandé d'utiliser *auto* comme valeur et de laisser *Flag* choisir une valeur appropriée.

Nouveau dans la version 3.6.

Comme avec *IntFlag*, si une combinaison de membres d'une classe *Flag* n'active aucune option, l'évaluation booléenne de la comparaison est *False* :

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color.0: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

Les options de base doivent avoir des puissances de deux pour valeurs (1, 2, 4, 8, ...) mais pas les combinaisons :

```
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```

Donner un nom à la valeur « aucune option activée » ne change pas sa valeur booléenne :

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

Note : Dans la plupart des cas, il est fortement recommandé d'utiliser *Enum* et *Flag* pour écrire du code nouveau, car *IntEnum* et *IntFlag* violent certains principes sémantiques d'une énumération (en pouvant être comparées à des entiers et donc, par transitivité, à d'autres énumérations). *IntEnum* et *IntFlag* ne doivent être utilisées que dans les cas où *Enum* et *Flag* ne suffisent pas ; par exemple quand des constantes entières sont remplacées par des énumérations, ou pour l'interopérabilité avec d'autres systèmes.

Autres

Bien que *IntEnum* fasse partie du module *enum*, elle serait très simple à implémenter hors de ce module :

```
class IntEnum(int, Enum):
    pass
```

Ceci montre comment définir des énumérations dérivées similaires ; par exemple une classe *StrEnum* qui dériverait de *str* au lieu de *int*.

Quelques règles :

1. Pour hériter de *Enum*, les types de mélange doivent être placés avant la classe *Enum* elle-même dans la liste des classes de base, comme dans l'exemple de *IntEnum* ci-dessus.
2. Même si une classe *Enum* peut avoir des membres de n'importe quel type, dès lors qu'un type de mélange est ajouté, tous les membres doivent être de ce type, p. ex. *int* ci-dessus. Cette restriction ne s'applique pas aux types de mélange qui ne font qu'ajouter des méthodes et ne définissent pas de type de données, tels *int* ou *str*.
3. Quand un autre type de données est mélangé, l'attribut *value* n'est pas identique au membre de l'énumération lui-même, bien qu'ils soient équivalents et égaux en comparaison.
4. Formatage de style *% : %s* et *%r* appellent respectivement les méthodes *__str__()* et *__repr__()* de la classe *Enum* ; les autres codes, comme *%i* ou *%h* pour *IntEnum*, s'appliquent au membre comme si celui-ci était converti en son type de mélange.

5. Chaînes de caractères formatées littérales : `str.format()` et `format()` appellent la méthode `__format__()` du type de mélange. Pour appeler les fonctions `str()` ou `repr()` de la classe `Enum`, il faut utiliser les codes de formatage `!s` ou `!r`.

8.13.14 Exemples intéressants

Bien que `Enum`, `IntEnum`, `IntFlag` et `Flag` soient conçues pour répondre à la majorité des besoins, elles ne peuvent répondre à tous. Voici quelques recettes d'énumération qui peuvent être réutilisées telles quelles, ou peuvent servir d'exemple pour développer vos propres énumérations.

Omettre les valeurs

Dans de nombreux cas, la valeur réelle de l'énumération n'a pas d'importance. Il y a plusieurs façons de définir ce type d'énumération simple :

- affecter des instances de `auto` aux valeurs
- affecter des instances de `object` aux valeurs
- affecter des chaînes de caractères aux valeurs pour les décrire
- affecter un n-uplet aux valeurs et définir une méthode `__new__()` pour remplacer les n-uplets avec un `int`

Utiliser une de ces méthodes indique à l'utilisateur que les valeurs n'ont pas d'importance. Cela permet aussi d'ajouter, de supprimer ou de ré-ordonner les membres sans avoir à ré-énumérer les membres existants.

Quelle que soit la méthode choisie, il faut fournir une méthode `repr()` qui masque les valeurs (pas importantes de toute façon) :

```
>>> class NoValue(Enum):
...     def __repr__(self):
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
... 
```

Avec auto

On utilise `auto` de la manière suivante :

```
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

Avec object

On utilise `object` de la manière suivante :

```
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

Avec une chaîne de caractères de description

On utilise une chaîne de caractères de la manière suivante :

```
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
'go'
```

Avec une méthode ad-hoc `__new__()`

On utilise une méthode `__new__()` d'énumération de la manière suivante :

```
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
2
```

Note : La méthode `__new__()`, si définie, est appelée à la création des membres de l'énumération ; elle est ensuite remplacée par la méthode `__new__()` de *Enum*, qui est utilisée après la création de la classe pour la recherche des membres existants.

OrderedEnum

Une énumération ordonnée qui n'est pas basée sur *IntEnum* et qui, par conséquent, respecte les invariants classiques de *Enum* (comme par exemple l'impossibilité de pouvoir être comparée à d'autres énumérations) :

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
```

(suite sur la page suivante)

(suite de la page précédente)

```

...     return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True

```

DuplicateFreeEnum

Lève une erreur si un membre est dupliqué, plutôt que de créer un alias :

```

>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum: %r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum: 'GRENE' --> 'GREEN'

```

Note : Cet exemple d'héritage de *Enum* est intéressant pour ajouter ou modifier des comportements comme interdire les alias. Si vous ne souhaitez qu'interdire les alias, il suffit d'utiliser le décorateur *unique()*.

Planet

Si `__new__()` ou `__init__()` sont définies, la valeur du membre de l'énumération sera passée à ces méthodes :

```
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27, 7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
...         self.mass = mass          # in kilograms
...         self.radius = radius      # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

8.13.15 En quoi les *Enums* sont différentes ?

Les *enums* ont une métaclass spéciale qui affecte de nombreux aspects des classes dérivées de *Enum* et de leur instances (membres).

Classes *Enum*

The `EnumMeta` metaclass is responsible for providing the `__contains__()`, `__dir__()`, `__iter__()` and other methods that allow one to do things with an *Enum* class that fail on a typical class, such as `list(Color)` or `some_var in Color`. `EnumMeta` is responsible for ensuring that various other methods on the final *Enum* class are correct (such as `__new__()`, `__getnewargs__()`, `__str__()` and `__repr__()`).

Membres d'Enum (c.-à-d. instances)

Il est intéressant de souligner que les membres d'une *Enum* sont des singletons. La classe `EnumMeta` les crée tous au moment de la création de la classe *Enum* elle-même et implémente une méthode `__new__()` spécifique. Cette méthode renvoie toujours les instances de membres déjà existantes pour être sûr de ne jamais en instancier de nouvelles.

Aspects approfondis

Noms de la forme `__dunder__` disponibles

`__members__` est une `OrderedDict` de correspondances `nom_du_membre` / `membre`. Elle n'est disponible que depuis la classe.

La méthode `__new__()`, si elle est définie, doit créer et renvoyer les membres de l'énumération ; affecter correctement l'attribut `_value_` du membre est également conseillé. Une fois que tous les membres ont été créés, cette méthode n'est plus utilisée.

Noms de la forme `_sunder_` disponibles

- `_name_` – nom du membre
- `_value_` – valeur du membre ; il est possible d'y accéder ou de la muer dans `__new__`
- `_missing_` – une fonction de recherche qui est appelée quand la valeur n'est pas trouvée ; elle peut être redéfinie
- `_order_` – utilisé en Python 2 ou 3 pour s'assurer que l'ordre des membres est cohérent (attribut de classe, supprimé durant la création de la classe)
- `_generate_next_value_` – utilisée par l'*API par fonction* et par *auto* pour obtenir une valeur appropriée à affecter à un membre de l'*enum* ; elle peut être redéfinie

Nouveau dans la version 3.6 : `_missing_`, `_order_`, `_generate_next_value_`

Pour faciliter la transition de Python 2 en Python 3, l'attribut `_order_` peut être défini. Il sera comparé au véritable ordre de l'énumération et lève une erreur si les deux ne correspondent pas :

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_
```

Note : En Python 2, l'attribut `_order_` est indispensable car l'ordre de la définition est perdu avant de pouvoir être enregistré.

Type des membres de `Enum`

Les membres de `Enum` sont des instances de leur classe `Enum`. On y accède normalement par `ClasseEnum.membre`. Dans certains cas, on peut également y accéder par `ClasseEnum.membre.membre`, mais ceci est fortement déconseillé car cette indirection est susceptible d'échouer, ou pire, de ne pas renvoyer le membre de la classe `Enum` désiré (c'est une autre bonne raison pour définir tous les noms des membres en majuscules) :

```
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
```

(suite sur la page suivante)

(suite de la page précédente)

```
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

Modifié dans la version 3.5.

Valeur booléenne des classes `Enum` et de leurs membres

Les membres d'une classe `Enum` mélangée avec un type non dérivé de `Enum` (comme `int`, `str`, etc.) sont évalués selon les règles du type de mélange. Sinon, tous les membres valent `True`. Pour faire dépendre l'évaluation booléenne de votre propre `Enum` de la valeur du membre, il faut ajouter le code suivant à votre classe :

```
def __bool__(self):
    return bool(self.value)
```

Les classes `Enum` valent toujours `True`.

Classes `Enum` avec des méthodes

Si votre classe `Enum` contient des méthodes supplémentaires, comme la classe `Planet` ci-dessus, elles s'afficheront avec un appel à `dir()` sur le membre, mais pas avec un appel sur la classe :

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__class__', '__doc__', '__members__', '__module__']
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

Combinaison de membres de `Flag`

Si une valeur issue de la combinaison de membres de `Flag` n'est pas associée explicitement à un membre, la fonction `repr()` inclut tous les membres et toutes les combinaisons de membres présents dans cette valeur :

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
...     CYAN = GREEN | BLUE
...
>>> Color(3) # named combination
<Color.YELLOW: 3>
>>> Color(7) # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>
```

Modules numériques et mathématiques

Les modules documentés dans ce chapitre fournissent des fonctions et des types autour des nombres et des mathématiques. Le module *numbers* définit une hiérarchie abstraite des types numériques. Les modules *math* et *cmath* contiennent des fonctions mathématiques pour les nombres à virgule flottante ou les nombres complexes. Le module *decimal* permet de représenter des nombres décimaux de manière exacte, et utilise une arithmétique de précision arbitraire.

Les modules suivants sont documentés dans ce chapitre :

9.1 *numbers* — Classes de base abstraites numériques

Code source : [Lib/numbers.py](#)

Le module *numbers* ([PEP 3141](#)) définit une hiérarchie de *classes de base abstraites* numériques qui définissent progressivement plus d'opérations. Aucun des types définis dans ce module ne peut être instancié.

class *numbers.Number*

La base de la hiérarchie numérique. Si vous voulez juste vérifier qu'un argument *x* est un nombre, peu importe le type, utilisez `isinstance(x, Number)`.

9.1.1 La tour numérique

class *numbers.Complex*

Les sous-classes de ce type décrivent des nombres complexes et incluent les opérations qui fonctionnent sur le type natif *complex*. Ce sont : les conversions vers *complex* et *bool*, *real*, *imag*, *+*, *-*, ***, */*, *abs()*, *conjugate()*, *==* et *!=*. Toutes sauf *-* et *!=* sont abstraites.

real

Abstrait. Récupère la partie réelle de ce nombre.

imag

Abstrait. Retrouve la partie imaginaire de ce nombre.

abstractmethod conjugate()

Abstrait. Renvoie le complexe conjugué. Par exemple, `(1+3j).conjugate() == (1-3j)`.

class numbers.Real

Real ajoute les opérations qui fonctionnent sur les nombres réels à *Complex*.

En bref, celles-ci sont : une conversion vers *float*, *math.trunc()*, *round()*, *math.floor()*, *math.ceil()*, *divmod()*, *//*, *%*, *<*, *<=*, *>* et *>=*.

Real fournit également des valeurs par défaut pour *complex()*, *real*, *imag* et *conjugate()*.

class numbers.Rational

Dérive *Real* et ajoute les propriétés *numerator* et *denominator* qui doivent être les plus petits possible. Avec celles-ci, il fournit une valeur par défaut pour *float()*.

numerator

Abstrait.

denominator

Abstrait.

class numbers.Integral

Dérive *Rational* et ajoute une conversion en *int*. Fournit des valeurs par défaut pour *float()*, *numerator* et *denominator*. Ajoute des méthodes abstraites pour **** et les opérations au niveau des bits : *<<*, *>>*, *&*, *^*, *|*, *~*.

9.1.2 Notes pour implémenter des types

Les développeurs doivent veiller à ce que des nombres égaux soient bien égaux lors de comparaisons et à ce qu'ils soient hachés aux mêmes valeurs. Cela peut être subtil s'il y a deux dérivations différentes des nombres réels. Par exemple, *fractions.Fraction* implémente *hash()* comme suit :

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

Ajouter plus d'ABC numériques

Il est bien entendu possible de créer davantage d'ABC pour les nombres et cette hiérarchie serait médiocre si elle excluait la possibilité d'en ajouter. Vous pouvez ajouter *MyFoo* entre *Complex* et *Real* ainsi :

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

Implémentation des opérations arithmétiques

Nous voulons implémenter les opérations arithmétiques de sorte que les opérations en mode mixte appellent une implémentation dont l'auteur connaît les types des deux arguments, ou convertissent chacun dans le type natif le plus proche et effectuent l'opération sur ces types. Pour les sous-types de *Integral*, cela signifie que `__add__()` et `__radd__()` devraient être définis comme suit :

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

Il existe 5 cas différents pour une opération de type mixte sur des sous-classes de *Complex*. Nous nous référerons à tout le code ci-dessus qui ne se réfère pas à *MyIntegral* et *OtherTypeIKnowAbout* comme « expression générique ». *a* est une instance de *A*, qui est un sous-type de *Complex* (*a* : *A* <: *Complex*) et *b* : *B* <: *Complex*. Considérons *a* + *b* :

1. Si *A* définit une `__add__()` qui accepte *b*, tout va bien.
2. Si *A* fait appel au code générique et que celui-ci renvoie une valeur de `__add__()`, nous manquons la possibilité que *B* définisse une `__radd__()` plus intelligent, donc le code générique devrait retourner *NotImplemented* dans `__add__()` (ou alors *A* ne doit pas implémenter `__add__()` du tout.)
3. Alors `__radd__()` de *B* a une chance. si elle accepte *a*, tout va bien.
4. Si elle fait appel au code générique, il n'y a plus de méthode possible à essayer, c'est donc ici que l'implémentation par défaut intervient.
5. Si *B* <: *A*, Python essaie *B*.`__radd__` avant *A*.`__add__`. C'est valable parce qu'elle est implémentée avec la connaissance de *A*, donc elle peut gérer ces instances avant de déléguer à *Complex*.

Si *A* <: *Complex* et *B* <: *Real* sans autre information, alors l'opération commune appropriée est celle impliquant *complex* et les deux `__radd__()` atterrissent à cet endroit, donc *a*+*b* == *b*+*a*.

Comme la plupart des opérations sur un type donné seront très similaires, il peut être utile de définir une fonction accessoire qui génère les instances résultantes et inverses d'un opérateur donné. Par exemple, *fractions.Fraction* utilise :

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
```

(suite sur la page suivante)

(suite de la page précédente)

```

        return fallback_operator(float(a), b)
    elif isinstance(b, complex):
        return fallback_operator(complex(a), b)
    else:
        return NotImplemented
forward.__name__ = '__' + fallback_operator.__name__ + '__'
forward.__doc__ = monomorphic_operator.__doc__

def reverse(b, a):
    if isinstance(a, Rational):
        # Includes ints.
        return monomorphic_operator(a, b)
    elif isinstance(a, numbers.Real):
        return fallback_operator(float(a), float(b))
    elif isinstance(a, numbers.Complex):
        return fallback_operator(complex(a), complex(b))
    else:
        return NotImplemented
reverse.__name__ = '_r' + fallback_operator.__name__ + '__'
reverse.__doc__ = monomorphic_operator.__doc__

return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...

```

9.2 Fonctions mathématiques — math

Ce module est toujours disponible. Il fournit l'accès aux fonctions mathématiques définies par le standard C.

Ces fonctions ne peuvent pas être utilisées avec les nombres complexes ; si vous avez besoin de la prise en charge des nombres complexes, utilisez les fonctions du même nom du module `cmath`. La séparation entre les fonctions qui gèrent les nombres complexes et les autres vient du constat que tous les utilisateurs ne souhaitent pas acquérir le niveau mathématique nécessaire à la compréhension des nombres complexes. Recevoir une exception plutôt qu'un nombre complexe en retour d'une fonction permet au programmeur de déterminer immédiatement comment et pourquoi ce nombre a été généré, avant que celui-ci ne soit passé involontairement en paramètre d'une autre fonction.

Les fonctions suivantes sont fournies dans ce module. Sauf mention contraire explicite, toutes les valeurs de retour sont des flottants.

9.2.1 Fonctions arithmétiques et de représentation

`math.ceil(x)`

Renvoie la partie entière par excès de x , le plus petit entier supérieur ou égal à x . Si x est un flottant, délègue à `x.__ceil()`, qui doit renvoyer une valeur *Integral*.

`math.copysign(x, y)`

Renvoie un flottant contenant la magnitude (valeur absolue) de x mais avec le signe de y . Sur les plates-formes prenant en charge les zéros signés, `copysign(1.0, -0.0)` renvoie `-1.0`.

`math.fabs(x)`

Renvoie la valeur absolue de x .

`math.factorial(x)`

Renvoie la factorielle de x . Lève une *ValueError* si x n'est pas entier ou s'il est négatif.

`math.floor(x)`

Renvoie la partie entière (par défaut) de x , le plus grand entier inférieur ou égal à x . Si x n'est pas un flottant, délègue à `x.__floor()`, qui doit renvoyer une valeur *Integral*.

`math.fmod(x, y)`

Renvoie `fmod(x, y)`, tel que défini par la bibliothèque C de la plate-forme. Notez que l'expression Python `x % y` peut ne pas renvoyer le même résultat. Le sens du standard C pour `fmod(x, y)` est d'être exactement (mathématiquement, à une précision infinie) égal à $x - n*y$ pour un entier n tel que le résultat a le signe de x et une magnitude inférieure à `abs(y)`. L'expression Python `x % y` renvoie un résultat avec le signe de y , et peut ne pas être calculable exactement pour des arguments flottants. Par exemple : `fmod(-1e-100, 1e100)` est `-1e-100`, mais le résultat de l'expression Python `-1e-100 % 1e100` est `1e100-1e-100`, qui ne peut pas être représenté exactement par un flottant et donc qui est arrondi à `1e100`. Pour cette raison, la fonction `fmod()` est généralement privilégiée quand des flottants sont manipulés, alors que l'expression Python `x % y` est privilégiée quand des entiers sont manipulés.

`math.frexp(x)`

Renvoie la mantisse et l'exposant de x dans un couple `(m, e)`. m est un flottant et e est un entier tels que `x == m * 2**e` exactement. Si x vaut zéro, renvoie `(0, 0)`, sinon `0.5 <= abs(m) < 1`. Ceci est utilisé pour « extraire » la représentation interne d'un flottant de manière portable.

`math.fsum(iterable)`

Renvoie une somme flottante exacte des valeurs dans l'itérable. Évite la perte de précision en gardant plusieurs sommes partielles intermédiaires :

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

La précision de cet algorithme dépend des garanties arithmétiques de IEEE-754 et des cas standards où le mode d'arrondi est *half-even*. Sur certaines versions non Windows, la bibliothèque C sous-jacente utilise une addition par précision étendue et peut occasionnellement effectuer un double-arrondi sur une somme intermédiaire causant la prise d'une mauvaise valeur du bit de poids faible.

Pour de plus amples discussions et deux approches alternatives, voir [ASPN cookbook recipes for accurate floating point summation](#).

`math.gcd(a, b)`

Renvoie le plus grand diviseur commun des entiers a et b . Si a ou b est différent de zéro, la valeur de `gcd(a, b)` est le plus grand entier positif qui divise à la fois a et b . `gcd(0, 0)` renvoie 0.

Nouveau dans la version 3.5.

`math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)`

Renvoie `True` si les valeurs a et b sont proches l'une de l'autre, et `False` sinon.

Déterminer si deux valeurs sont considérées comme « proches » se fait à l'aide des tolérances absolues et relatives passées en paramètres.

`rel_tol` est la tolérance relative — c'est la différence maximale permise entre *a* et *b*, relativement à la plus grande valeur de *a* ou de *b*. Par exemple, pour définir une tolérance de 5%, précisez `rel_tol=0.05`. La tolérance par défaut est `1e-09`, ce qui assure que deux valeurs sont les mêmes à partir de la 9^e décimale. `rel_tol` doit être supérieur à zéro.

`abs_tol` est la tolérance absolue minimale — utile pour les comparaisons proches de zéro. `abs_tol` doit valoir au moins zéro.

Si aucune erreur n'est rencontrée, le résultat sera : `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

Les valeurs spécifiques suivantes : `NaN`, `inf`, et `-inf` définies dans la norme IEEE 754 seront manipulées selon les règles du standard IEEE. En particulier, `NaN` n'est considéré proche d'aucune autre valeur, `NaN` inclus. `inf` et `-inf` ne sont considérées proches que d'elles-mêmes.

Nouveau dans la version 3.5.

Voir aussi :

PEP 485 — Une fonction pour tester des quasi-égalités

`math.isfinite(x)`

Renvoie `True` si *x* n'est ni infini, ni `NaN`, et `False` sinon. (Notez que `0.0` est considéré comme fini.)

Nouveau dans la version 3.2.

`math.isinf(x)`

Renvoie `True` si *x* vaut l'infini positif ou négatif, et `False` sinon.

`math.isnan(x)`

Renvoie `True` si *x* est `NaN` (*Not a Number*, ou *Pas un Nombre* en français), et `False` sinon.

`math.ldexp(x, i)`

Renvoie `x * (2**i)`. C'est essentiellement l'inverse de la fonction `frexp()`.

`math.modf(x)`

Renvoie les parties entière et fractionnelle de *x*. Les deux résultats ont le signe de *x* et sont flottants.

`math.trunc(x)`

Renvoie la valeur *Real* *x* tronquée en un *Integral* (habituellement un entier). Délègue à `x.__trunc__()`.

Notez que les fonctions `frexp()` et `modf()` ont un système d'appel différent de leur homologue C : elles prennent un seul argument et renvoient une paire de valeurs au lieu de placer la seconde valeur de retour dans un *paramètre de sortie* (cela n'existe pas en Python).

Pour les fonctions `ceil()`, `floor()`, et `modf()`, notez que *tous* les nombres flottants de magnitude suffisamment grande sont des entiers exacts. Les flottants de Python n'ont généralement pas plus de 53 *bits* de précision (tels que le type C `double` de la plate-forme), en quel cas tout flottant *x* tel que `abs(x) >= 2**52` n'a aucun *bit* fractionnel.

9.2.2 Fonctions logarithme et exponentielle

`math.exp(x)`

Renvoie `e**x`.

`math.expm1(x)`

Renvoie `e**x - 1`. Pour de petits flottants, la soustraction `exp(x) - 1` peut résulter en une *perte significative de précision* ; la fonction `expm1()` fournit un moyen de calculer cette quantité en précision complète :

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1 # gives result accurate to 11 places
1.0000050000069649e-05
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> expm1(1e-5)      # result accurate to full precision
1.0000050000166668e-05
```

Nouveau dans la version 3.2.

`math.log(x[, base])`

Avec un argument, renvoie le logarithme naturel de x (en base e).

Avec deux arguments, renvoie le logarithme de x en la *base* donnée, calculé par $\log(x) / \log(\text{base})$.

`math.log1p(x)`

Renvoie le logarithme naturel de $1+x$ (en base e). Le résultat est calculé par un moyen qui reste exact pour x proche de zéro.

`math.log2(x)`

Renvoie le logarithme en base 2 de x . C'est en général plus précis que $\log(x, 2)$.

Nouveau dans la version 3.3.

Voir aussi :

`int.bit_length()` renvoie le nombre de bits nécessaires pour représenter un entier en binaire, en excluant le signe et les zéros de début.

`math.log10(x)`

Renvoie le logarithme de x en base 10. C'est habituellement plus exact que $\log(x, 10)$.

`math.pow(x, y)`

Renvoie x élevé à la puissance y . Les cas exceptionnels suivent l'annexe "F" du standard C99 autant que possible. En particulier, `pow(1.0, x)` et `pow(x, 0.0)` renvoient toujours 1.0, même si x est zéro ou NaN. Si à la fois x et y sont finis, x est négatif et y n'est pas entier, alors `pow(x, y)` est non défini et lève une `ValueError`. À l'inverse de l'opérateur interne `**`, la fonction `math.pow()` convertit ses deux arguments en *float*. Utilisez `**` ou la primitive `pow()` pour calculer des puissances exactes d'entiers.

`math.sqrt(x)`

Renvoie la racine carrée de x .

9.2.3 Fonctions trigonométriques

`math.acos(x)`

Renvoie l'arc cosinus de x , en radians.

`math.asin(x)`

Renvoie l'arc sinus de x , en radians.

`math.atan(x)`

Renvoie l'arc tangente de x , en radians.

`math.atan2(y, x)`

Renvoie $\text{atan}(y / x)$, en radians. Le résultat est entre $-\pi$ et π . Le vecteur du plan allant de l'origine vers le point (x, y) forme cet angle avec l'axe X positif. L'intérêt de `atan2()` est que le signe des deux entrées est connu. Donc elle peut calculer le bon quadrant pour l'angle. par exemple `atan(1)` et `atan2(1, 1)` donnent tous deux $\pi/4$, mais `atan2(-1, -1)` donne $-3\pi/4$.

`math.cos(x)`

Renvoie le cosinus de x radians.

`math.hypot(x, y)`

Renvoie la norme euclidienne $\sqrt{x^2 + y^2}$. C'est la longueur du vecteur allant de l'origine au point (x, y) .

`math.sin(x)`
Renvoie le sinus de x radians.

`math.tan(x)`
Renvoie la tangente de x radians.

9.2.4 Conversion angulaire

`math.degrees(x)`
Convertit l'angle x de radians en degrés.

`math.radians(x)`
Convertit l'angle x de degrés en radians.

9.2.5 Fonctions hyperboliques

Les **fonctions hyperboliques** sont analogues à des fonctions trigonométriques qui sont basées sur des hyperboles au lieu de cercles.

`math.acosh(x)`
Renvoie l'arc cosinus hyperbolique de x .

`math.asinh(x)`
Renvoie l'arc sinus hyperbolique de x .

`math.atanh(x)`
Renvoie l'arc tangente hyperbolique de x .

`math.cosh(x)`
Renvoie le cosinus hyperbolique de x .

`math.sinh(x)`
Renvoie le sinus hyperbolique de x .

`math.tanh(x)`
Renvoie la tangente hyperbolique de x .

9.2.6 Fonctions spéciales

`math.erf(x)`
Renvoie la **fonction d'erreur** en x .
La fonction `erf()` peut être utilisée pour calculer des fonctions statistiques usuelles telles que la **répartition de la loi normale** :

```
def phi(x):  
    'Cumulative distribution function for the standard normal distribution'  
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

Nouveau dans la version 3.2.

`math.erfc(x)`
Renvoie la fonction d'erreur complémentaire en x . La **fonction d'erreur complémentaire** est définie par $1.0 - \text{erf}(x)$. Elle est utilisée pour les grandes valeurs de x , où la soustraction en partant de 1,0 entraînerait une **perte de précision**.
Nouveau dans la version 3.2.

`math.gamma(x)`

Renvoie la fonction [Gamma](#) en x .

Nouveau dans la version 3.2.

`math.lgamma(x)`

Renvoie le logarithme naturel de la valeur absolue de la fonction gamma en x .

Nouveau dans la version 3.2.

9.2.7 Constantes

`math.pi`

La constante mathématique $\pi = 3.141592\dots$, à la précision disponible.

`math.e`

La constante mathématique $e = 2.718281\dots$, à la précision disponible.

`math.tau`

La constante mathématique $\tau = 6.283185\dots$, à la précision disponible. Tau est une constante du cercle égale à 2π , le ratio entre la circonférence d'un cercle et son rayon. Pour en apprendre plus sur Tau, regardez la vidéo de Vi Hart, [Pi is \(still\) Wrong](#), et profitez-en (ou pas) pour célébrer (ou pas) le [Jour de Tau](#) en mangeant (ou pas) deux fois plus camembert. (Y'a un jeu de mot, en anglais, avec le *Tau day*, c'est *eating twice as much pie!* (« deux fois plus de π » ?)).

Nouveau dans la version 3.6.

`math.inf`

Un flottant positif infini. (Pour un infini négatif, utilisez `-math.inf`.) Équivalent au résultat de `float('inf')`.

Nouveau dans la version 3.5.

`math.nan`

Un flottant valant NaN. Équivalent au résultat de `float('nan')`.

Nouveau dans la version 3.5.

CPython implementation detail : Le module `math` consiste majoritairement en un conteneur pour les fonctions mathématiques de la bibliothèque C de la plate-forme. Le comportement dans les cas spéciaux suit l'annexe “F” du standard C99 quand c'est approprié. L'implémentation actuelle lève une `ValueError` pour les opérations invalides telles que `sqrt(-1.0)` ou `log(0.0)` (où le standard C99 recommande de signaler que l'opération est invalide ou qu'il y a division par zéro), et une `OverflowError` pour les résultats qui débordent (par exemple `exp(1000.0)`). `NaN` ne sera renvoyé pour aucune des fonctions ci-dessus, sauf si au moins un des arguments de la fonction vaut `NaN`. Dans ce cas, la plupart des fonctions renvoient `NaN`, mais (à nouveau, selon l'annexe “F” du standard C99) il y a quelques exceptions à cette règle, par exemple `pow(float('nan'), 0.0)` ou `hypot(float('nan'), float('inf'))`.

Notez que Python ne fait aucun effort pour distinguer les NaNs signalétiques des NaNs silencieux, et le comportement de signalement des NaNs reste non-spécifié. Le comportement standard est de traiter tous les NaNs comme s'ils étaient silencieux.

Voir aussi :

Module `cmath` Version complexe de beaucoup de ces fonctions.

9.3 Fonctions mathématiques pour nombres complexes — `cmath`

Ce module est toujours disponible. Il fournit l'accès aux fonctions mathématiques pour les nombres complexes. Les fonctions de ce module acceptent les entiers, les nombres flottants ou les nombres complexes comme arguments. Elles accepteront également tout objet Python ayant une méthode `__complex__()` ou `__float__()` : ces méthodes sont utilisées pour convertir l'objet en un nombre complexe ou respectivement un nombre flottant, et la fonction est ensuite appliquée sur le résultat de la conversion.

Note : Sur les plate-formes avec un support système et matériel des zéros signés, les fonctions incluant une coupure complexe sont continues *de chaque* côté de la coupure : le signe du zéro distingue les deux extrémités de la coupure. Sur les plate-formes ne supportant pas les zéros signés, la continuité est spécifiée en-dessous.

9.3.1 Conversion vers et à partir de coordonnées polaires

Un nombre complexe Python `z` est stocké de manière interne en coordonnées *cartésiennes*. Il est entièrement défini par sa *partie réelle* `z.real` et sa *partie complexe* `z.imag`. En d'autres termes :

```
z == z.real + z.imag*1j
```

Les *coordonnées polaires* donnent une manière alternative de représenter un nombre complexe. En coordonnées polaires, un nombre complexe `z` est défini par son module `r` et par son argument (*angle de phase*) `phi`. Le module `r` est la distance entre `z` et l'origine, alors que l'argument `phi` est l'angle (dans le sens inverse des aiguilles d'une montre, ou sens trigonométrique), mesuré en radians, à partir de l'axe X positif, et vers le segment de droite joignant `z` à l'origine.

Les fonctions suivantes peuvent être utilisées pour convertir à partir des coordonnées rectangulaires natives vers les coordonnées polaires, et vice-versa.

`cmath.phase(x)`

Renvoie l'argument de `x`, dans un nombre flottant. `phase(x)` est équivalent à `math.atan2(x.imag, x.real)`. Le résultat se situe dans l'intervalle $[-\pi, +\pi]$, et la coupure par cette opération se situe sur l'axe X négatif, continue par au-dessus. Sur les systèmes supportant les zéros signés (ce qui inclut la plupart des systèmes utilisés actuellement), cela signifie que le signe du résultat est le même que `x.imag` même quand `x.imag` vaut zéro :

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

Note : Le module (valeur absolue) d'un nombre complexe `x` peut être calculé en utilisant la primitive `abs()`. Il n'y a pas de fonction spéciale du module `cmath` pour cette opération.

`cmath.polar(x)`

Renvoie la représentation de `x` en coordonnées polaires. Renvoie une paire `(r, phi)` où `r` est le module de `x` et `phi` est l'argument de `x`. `polar(x)` est équivalent à `(abs(x), phase(x))`.

`cmath.rect(r, phi)`

Renvoie le nombre complexe `x` dont les coordonnées polaires sont `r` et `phi`. Équivalent à `r * (math.cos(phi) + math.sin(phi)*1j)`.

9.3.2 Fonctions logarithme et exponentielle

`cmath.exp(x)`

Renvoie la valeur exponentielle e^{**x} .

`cmath.log(x[, base])`

Renvoie le logarithme de x dans la *base* précisée. Si la *base* n'est pas précisée, le logarithme *naturel* (népérien) de x est renvoyé. Il y a une coupure, partant de 0 sur l'axe réel négatif et vers $-\infty$, continue par au-dessus.

`cmath.log10(x)`

Renvoie le logarithme en base 10 de x . Elle a la même coupure que `log()`.

`cmath.sqrt(x)`

Renvoie la racine carrée de x . Elle a la même coupure que `log()`.

9.3.3 Fonctions trigonométriques

`cmath.acos(x)`

Renvoie l'arc cosinus de x . Il y a deux coupures : une allant de 1 sur l'axe réel vers ∞ , continue par en-dessous ; l'autre allant de -1 sur l'axe réel vers $-\infty$, continue par au-dessus.

`cmath.asin(x)`

Renvoie l'arc sinus de x . Elle a les mêmes coupures que `acos()`.

`cmath.atan(x)`

Renvoie la tangente de x . Il y a deux coupures : une allant de $1j$ sur l'axe imaginaire vers ∞j , continue par la droite ; l'autre allant de $-1j$ sur l'axe imaginaire vers $-\infty j$, continue par la gauche.

`cmath.cos(x)`

Renvoie le cosinus de x .

`cmath.sin(x)`

Renvoie le sinus de x .

`cmath.tan(x)`

Renvoie la tangente de x .

9.3.4 Fonctions hyperboliques

`cmath.acosh(x)`

Renvoie l'arc cosinus hyperbolique de x . Il y a une coupure, allant de 1 sur l'axe réel vers $-\infty$, continue par au-dessus.

`cmath.asinh(x)`

Renvoie l'arc sinus hyperbolique de x . Il y a deux coupures : une allant de $1j$ sur l'axe imaginaire vers ∞j , continue par la droite ; l'autre allant de $-1j$ sur l'axe imaginaire vers ∞j , continue par la gauche.

`cmath.atanh(x)`

Renvoie l'arc tangente hyperbolique de x . Il y a deux coupures : une allant de 1 sur l'axe réel allant vers ∞ , continue par en-dessous ; l'autre allant de -1 sur l'axe réel vers $-\infty$, continue par au-dessus.

`cmath.cosh(x)`

Renvoie le cosinus hyperbolique de x .

`cmath.sinh(x)`

Renvoie le sinus hyperbolique de x .

`cmath.tanh(x)`

Renvoie la tangente hyperbolique de x .

9.3.5 Fonctions de classifications

`cmath.isfinite(x)`

Renvoie `True` si la partie réelle *et* la partie imaginaire de *x* sont finies, et `False` sinon.

Nouveau dans la version 3.2.

`cmath.isinf(x)`

Renvoie `True` si soit la partie réelle *ou* la partie imaginaire de *x* est infinie, et `False` sinon.

`cmath.isnan(x)`

Renvoie `True` si soit la partie réelle *ou* la partie imaginaire de *x* est NaN, et `False` sinon.

`cmath.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)`

Renvoie `True` si les valeurs *a* et *b* sont proches l'une de l'autre, et `False` sinon.

Déterminer si deux valeurs sont proches se fait à l'aide des tolérances absolues et relatives données en paramètres. *rel_tol* est la tolérance relative – c'est la différence maximale permise entre *a* et *b*, relativement à la plus grande valeur de *a* ou de *b*. Par exemple, pour définir une tolérance de 5%, précisez *rel_tol*=0.05. La tolérance par défaut est 1e-09, ce qui assure que deux valeurs sont les mêmes à partir de la 9^e décimale. *rel_tol* doit être supérieur à zéro.

abs_tol est la tolérance absolue minimale – utile pour les comparaisons proches de zéro. *abs_tol* doit valoir au moins zéro.

Si aucune erreur n'est rencontrée, le résultat sera : `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

Les valeurs spécifiques suivantes : NaN, `inf`, et `-inf` définies dans la norme IEEE 754 seront manipulées selon les règles du standard IEEE. En particulier, NaN n'est considéré proche d'aucune autre valeur, NaN inclus. `inf` et `-inf` ne sont considérés proches que d'eux-mêmes.

Nouveau dans la version 3.5.

Voir aussi :

PEP 485 – Une fonction pour tester des égalités approximées

9.3.6 Constantes

`cmath.pi`

La constante mathématique π , en tant que flottant.

`cmath.e`

La constante mathématique *e*, en tant que flottant.

`cmath.tau`

La constante mathématique τ , sous forme de flottant.

Nouveau dans la version 3.6.

`cmath.inf`

Nombre à virgule flottante positif infini. Équivaut à `float('inf')`.

Nouveau dans la version 3.6.

`cmath.infj`

Nombre complexe dont la partie réelle vaut zéro et la partie imaginaire un infini positif. Équivalent à `complex(0, float('inf'))`.

Nouveau dans la version 3.6.

`cmath.nan`

Un nombre à virgule flottante NaN (*Not a number*). Équivalent à `float('nan')`.

Nouveau dans la version 3.6.

`cmath.nanj`

Nombre complexe dont la partie réelle vaut zéro et la partie imaginaire vaut un *NaN*. Équivalent à `complex(0.0, float('nan'))`.

Nouveau dans la version 3.6.

Notez que la sélection de fonctions est similaire, mais pas identique, à celles du module `math`. La raison d'avoir deux modules est que certains utilisateurs ne sont pas intéressés par les nombres complexes, et peut-être ne savent même pas ce qu'ils sont. Ils préféreraient alors que `math.sqrt(-1)` lève une exception au lieu de renvoyer un nombre complexe. Également, notez que les fonctions définies dans `cmath` renvoient toujours un nombre complexe, même si le résultat peut être exprimé à l'aide d'un nombre réel (en quel cas la partie imaginaire du complexe vaut zéro).

Une note sur les *coupures* : ce sont des courbes sur lesquelles la fonction n'est pas continue. Ce sont des caractéristiques nécessaires de beaucoup de fonctions complexes. Il est supposé que si vous avez besoin d'utiliser des fonctions complexes, vous comprendrez ce que sont les coupures. Consultez n'importe quel livre (pas trop élémentaire) sur les variables complexes pour plus d'informations. Pour des informations sur les choix des coupures à des fins numériques, voici une bonne référence :

Voir aussi :

Kahan, W : Branch cuts for complex elementary functions; or, Much ado about nothing's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165–211.

9.4 decimal — Arithmétique décimale en virgule fixe et flottante

Code source : [Lib/decimal.py](#)

Le module `decimal` fournit une arithmétique en virgule flottante rapide et produisant des arrondis mathématiquement corrects. Il possède plusieurs avantages en comparaison au type `float` :

- Le module `decimal` « est basé sur un modèle en virgule flottante conçu pour les humains, qui suit ce principe directeur : l'ordinateur doit fournir un modèle de calcul qui fonctionne de la même manière que le calcul qu'on apprend à l'école » – extrait (traduit) de la spécification de l'arithmétique décimale.
- Les nombres décimaux peuvent être représentés exactement en base décimale flottante. En revanche, des nombres tels que 1.1 ou 1.2 n'ont pas de représentation exacte en base binaire flottante. L'utilisateur final ne s'attend typiquement pas à obtenir 3.3000000000000003 lorsqu'il saisit 1.1 + 2.2, ce qui se passe en arithmétique binaire à virgule flottante.
- Ces inexactitudes ont des conséquences en arithmétique. En base décimale à virgule flottante, 0.1 + 0.1 + 0.1 - 0.3 est exactement égal à zéro. En virgule flottante binaire, l'ordinateur l'évalue à 5.5511151231257827e-017. Bien que très proche de zéro, cette différence induit des erreurs lors des tests d'égalité, erreurs qui peuvent s'accumuler. Pour ces raisons `decimal` est le module utilisé pour des applications comptables ayant des contraintes strictes de fiabilité.
- Le module `decimal` incorpore la notion de chiffres significatifs, tels que 1.30 + 1.20 est égal à 2.50. Le dernier zéro n'est conservé que pour respecter le nombre de chiffres significatifs. C'est également l'affichage préféré pour représenter des sommes d'argent. Pour la multiplication, l'approche « scolaire » utilise tout les chiffres présents dans les facteurs. Par exemple, 1.3 * 1.2 donnerait 1.56 tandis que 1.30 * 1.20 donnerait 1.5600.
- Contrairement à l'arithmétique en virgule flottante binaire, le module `decimal` possède un paramètre de précision ajustable (par défaut à 28 chiffres significatifs) qui peut être aussi élevée que nécessaire pour un problème donné :

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571428571429')
```

- L'arithmétique binaire et décimale en virgule flottante sont implémentées selon des standards publiés. Alors que le type `float` n'expose qu'une faible portion de ses capacités, le module `decimal` expose tous les composants nécessaires du standard. Lorsque nécessaire, le développeur a un contrôle total de la gestion de signal et de l'arrondi. Cela inclut la possibilité de forcer une arithmétique exacte en utilisant des exceptions pour bloquer toute opération inexacte.
- Le module `decimal` a été conçu pour gérer « sans préjugé, à la fois une arithmétique décimale non-arrondie (aussi appelée arithmétique en virgule fixe) et à la fois une arithmétique en virgule flottante. » (extrait traduit de la spécification de l'arithmétique décimale).

Le module est conçu autour de trois concepts : le nombre décimal, le contexte arithmétique et les signaux.

Un `Decimal` est immuable. Il a un signe, un coefficient, et un exposant. Pour préserver le nombre de chiffres significatifs, les zéros en fin de chaîne ne sont pas tronqués. Les décimaux incluent aussi des valeurs spéciales telles que `Infinity`, `-Infinity`, et `NaN`. Le standard fait également la différence entre `-0` et `+0`.

Le contexte de l'arithmétique est un environnement qui permet de configurer une précision, une règle pour l'arrondi, des limites sur l'exposant, des options indiquant le résultat des opérations et si les signaux (remontés lors d'opérations illégales) sont traités comme des exceptions Python. Les options d'arrondi incluent `ROUND_CEILING`, `ROUND_DOWN`, `ROUND_FLOOR`, `ROUND_HALF_DOWN`, `ROUND_HALF_EVEN`, `ROUND_HALF_UP`, `ROUND_UP`, et `ROUND_05UP`.

Les signaux sont des groupes de conditions exceptionnelles qui surviennent durant le calcul. Selon les besoins de l'application, les signaux peuvent être ignorés, considérés comme de l'information, ou bien traités comme des exceptions. Les signaux dans le module `decimal` sont : `Clamped`, `InvalidOperation`, `DivisionByZero`, `Inexact`, `Rounded`, `Subnormal`, `Overflow`, `Underflow` et `FloatOperation`.

Chaque signal est configurable indépendamment. Quand une opération illégale survient, le signal est mis à 1, puis s'il est configuré pour, une exception est levée. La mise à 1 est persistante, l'utilisateur doit donc les remettre à zéro avant de commencer un calcul qu'il souhaite surveiller.

Voir aussi :

- La spécification d'IBM sur l'arithmétique décimale : [The General Decimal Arithmetic Specification](#).

9.4.1 Introduction pratique

Commençons par importer le module, regarder le contexte actuel avec `getcontext()`, et si nécessaire configurer la précision, l'arrondi, et la gestion des signaux :

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7           # Set a new precision
```

Les instances de `Decimal` peuvent être construites avec des `int`, des `str`, des `floats` ou des `tuples`. La construction depuis un entier ou un `float` effectue la conversion exacte de cet entier ou de ce `float`. Les nombres décimaux incluent des valeurs spéciales telles que `NaN` qui signifie en anglais « *Not a number* », en français « pas un nombre », des `Infinity` positifs ou négatifs et `-0` :

```
>>> getcontext().prec = 28
>>> Decimal(10)
```

(suite sur la page suivante)

(suite de la page précédente)

```
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.1400000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.414213562373095048801688724')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

Si un signal *FloatOperation* est détecté, un mélange accidentel d'objets *Decimal* et de *float* dans les constructeurs ou des opérations de comparaisons, une exception est levée :

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [
>>> Decimal('3.5') == 3.5
True
```

Nouveau dans la version 3.3.

Le nombre de chiffres significatifs d'un nouvel objet `Decimal` est déterminé entièrement par le nombre de chiffres saisis. La précision et les règles d'arrondis n'interviennent que lors d'opérations arithmétiques.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

Si les limites internes de la version en C sont dépassées, la construction d'un objet décimal lève l'exception *InvalidOperation*:

```
>>> Decimal("1e999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [
```

Modifié dans la version 3.3.

Les objets `Decimal` interagissent très bien avec le reste de Python. Voici quelques exemple d'opérations avec des décimaux :

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

Et certaines fonctions mathématiques sont également disponibles sur des instances de `Decimal` :

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

La méthode `quantize()` arrondit un nombre à un exposant fixe. Cette méthode est utile pour des applications monétaires qui arrondissent souvent un résultat à un nombre de chiffres significatifs exact :

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

Comme montré plus haut, la fonction `getcontext()` accède au contexte actuel et permet de modifier les paramètres. Cette approche répond aux besoins de la plupart des applications.

Pour un travail plus avancé, il peut être utile de créer des contextes alternatifs en utilisant le constructeur de `Context`. Pour activer cet objet `Context`, utilisez la fonction `setcontext()`.

En accord avec le standard, le module `decimal` fournit des objets `Context` standards, `BasicContext` et `ExtendedContext`. Le premier est particulièrement utile pour le débogage car beaucoup des pièges sont activés dans cet objet.

```

>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
       capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0

```

Les objets `Context` ont aussi des options pour détecter des opérations illégales lors des calculs. Ces options restent activées jusqu'à ce qu'elles soit remises à zéro de manière explicite. Il convient donc de remettre à zéro ces options avant chaque inspection de chaque calcul, avec la méthode `clear_flags()`.

```

>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
       capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])

```

Les options montrent que l'approximation de π par une fraction a été arrondie (les chiffres au delà de la précision spécifiée par l'objet `Context` ont été tronqués) et que le résultat est différent (certains des chiffres tronqués étaient différents de zéro).

L'activation des pièges se fait en utilisant un dictionnaire dans l'attribut `traps` de l'objet `Context` :

```

>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0

```

La plupart des applications n'ajustent l'objet `Context` qu'une seule fois, au démarrage. Et, dans beaucoup d'applications, les données sont convertie une fois pour toutes en *Decimal*. Une fois le `Context` initialisé, et les objets `Decimal` créés, l'essentiel du programme manipule la donnée de la même manière qu'avec les autres types numériques Python.

9.4.2 Les objets Decimal

class decimal.Decimal (*value*="0", *context*=None)

Construire un nouvel objet *Decimal* à partir de *value*.

value peut être un entier, une chaîne de caractères, un tuple, *float*, ou une autre instance de *Decimal*. Si *value* n'est pas fourni, le constructeur renvoie `Decimal('0')`. Si *value* est une chaîne de caractère, elle doit correspondre à la syntaxe décimale en dehors des espaces de début et de fin, ou des tirets bas, qui sont enlevés :

```
sign          ::= '+' | '-'
digit         ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator     ::= 'e' | 'E'
digits        ::= digit [digit]...
decimal-part  ::= digits '.' [digits] | ['.' ] digits
exponent-part ::= indicator [sign] digits
infinity      ::= 'Infinity' | 'Inf'
nan           ::= 'NaN' [digits] | 'sNaN' [digits]
numeric-value ::= decimal-part [exponent-part] | infinity
numeric-string ::= [sign] numeric-value | [sign] nan
```

Les chiffres codés en Unicode sont aussi autorisés, là où *digit* apparaît. Cela inclut des chiffres décimaux venant d'autres alphabets (par exemple les chiffres indo-arabes ou Devanagari) ainsi que les chiffres de pleine largeur `'\uff10'` jusqu'à `'\uff19'`.

Si *value* est un *tuple*, il doit avoir 3 éléments, le signe (0 pour positif ou 1 pour négatif), un *tuple* de chiffres, et un entier représentant l'exposant. Par exemple, `Decimal((0, (1, 4, 1, 4), -3))` construit l'objet `Decimal('1.414')`.

Si *value* est un *float*, la valeur en binaire flottant est convertie exactement à son équivalent décimal. Cette conversion peut parfois nécessiter 53 chiffres significatifs ou plus. Par exemple, `Decimal(float('1.1'))` devient `Decimal('1.100000000000000088817841970012523233890533447265625')`.

La précision spécifiée dans *Context* n'affecte pas le nombre de chiffres stockés. Cette valeur est déterminée exclusivement par le nombre de chiffres dans *value*. Par exemple, `Decimal('3.00000')` enregistre les 5 zéros même si la précision du contexte est de 3.

L'objectif de l'argument *context* est de déterminer ce que Python doit faire si *value* est une chaîne avec un mauvais format. Si l'option *InvalidOperation* est activée, une exception est levée, sinon le constructeur renvoie un objet *Decimal* avec la valeur NaN.

Une fois construit, les objets *Decimal* sont immuables.

Modifié dans la version 3.2 : L'argument du constructeur peut désormais être un objet *float*.

Modifié dans la version 3.3 : Un argument *float* lève une exception si l'option *FloatOperation* est activé. Par défaut l'option ne l'est pas.

Modifié dans la version 3.6 : Les tirets bas sont autorisés pour regrouper, tout comme pour l'arithmétique en virgule fixe et flottante.

Les objets *Decimal* partagent beaucoup de propriétés avec les autres types numériques natifs tels que *float* et *int*. Toutes les opérations mathématiques et méthodes sont conservées. De même les objets *Decimal* peuvent être copiés, sérialisés via le module *pickle*, affichés, utilisés comme clé de dictionnaire, éléments d'ensembles, comparés, classés, et convertis vers un autre type (tel que *float* ou *int*).

Il existe quelques différences mineures entre l'arithmétique entre les objets décimaux et l'arithmétique avec les entiers et les *float*. Quand l'opérateur modulo `%` est appliqué sur des objets décimaux, le signe du résultat est le signe du *dividend* plutôt que le signe du diviseur :

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

L'opérateur division entière, `//` se comporte de la même manière, retournant la partie entière du quotient, plutôt que son arrondi, de manière à préserver l'identité d'Euclide $x == (x // y) * y + x \% y$:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

Les opérateurs `//` et `%` implémentent la division entière et le reste (ou modulo), respectivement, tel que décrit dans la spécification.

Les objets `Decimal` ne peuvent généralement pas être combinés avec des `float` ou des objets `Fraction`. `Fraction` lors d'opérations arithmétiques : toute addition entre un `Decimal` avec un `float`, par exemple, lève une exception `TypeError`. Cependant, il est possible d'utiliser les opérateurs de comparaison entre instances de `Decimal` avec les autres types numériques. Cela évite d'avoir des résultats absurdes lors des tests d'égalité entre différents types.

Modifié dans la version 3.2 : Les comparaisons inter-types entre `Decimal` et les autres types numériques sont désormais intégralement gérés.

In addition to the standard numeric properties, decimal floating point objects also have a number of specialized methods :

adjusted()

Return the adjusted exponent after shifting out the coefficient's rightmost digits until only the lead digit remains : `Decimal('321e+5').adjusted()` returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

as_integer_ratio()

Return a pair (n, d) of integers that represent the given `Decimal` instance as a fraction, in lowest terms and with a positive denominator :

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

La conversion est exacte. Lève une `OverflowError` sur l'infini et `ValueError` sur les NaN's.

Nouveau dans la version 3.6.

as_tuple()

Return a *named tuple* representation of the number : `DecimalTuple(sign, digits, exponent)`.

canonical()

Return the canonical encoding of the argument. Currently, the encoding of a `Decimal` instance is always canonical, so this operation returns its argument unchanged.

compare(other, context=None)

Compare the values of two `Decimal` instances. `compare()` returns a `Decimal` instance, and if either operand is a NaN then the result is a NaN :

```
a or b is a NaN ==> Decimal('NaN')
a < b           ==> Decimal('-1')
a == b          ==> Decimal('0')
a > b           ==> Decimal('1')
```

compare_signal(other, context=None)

This operation is identical to the `compare()` method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

compare_total(other, context=None)

Compare two operands using their abstract representation rather than their numerical value. Similar to the `compare()` method, but the result gives a total ordering on `Decimal` instances. Two `Decimal` instances with the same numeric value but different representations compare unequal in this ordering :

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is `Decimal('0')` if both operands have the same representation, `Decimal('-1')` if the first operand

is lower in the total order than the second, and `Decimal('1')` if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

This operation is unaffected by context and is quiet : no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

compare_total_mag (*other*, *context=None*)

Compare two operands using their abstract representation rather than their value as in `compare_total()`, but ignoring the sign of each operand. `x.compare_total_mag(y)` is equivalent to `x.copy_abs().compare_total(y.copy_abs())`.

This operation is unaffected by context and is quiet : no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

conjugate ()

Just returns self, this method is only to comply with the Decimal Specification.

copy_abs ()

Return the absolute value of the argument. This operation is unaffected by the context and is quiet : no flags are changed and no rounding is performed.

copy_negate ()

Return the negation of the argument. This operation is unaffected by the context and is quiet : no flags are changed and no rounding is performed.

copy_sign (*other*, *context=None*)

Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example :

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by context and is quiet : no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

exp (*context=None*)

Return the value of the (natural) exponential function e^{**x} at the given number. The result is correctly rounded using the `ROUND_HALF_EVEN` rounding mode.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

from_float (*f*)

Classmethod that converts a float to a decimal number, exactly.

Note `Decimal.from_float(0.1)` is not the same as `Decimal("0.1")`. Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is $0x1.999999999999ap-4$. That equivalent value in decimal is `0.1000000000000000055511151231257827021181583404541015625`.

Note : From Python 3.2 onwards, a `Decimal` instance can also be constructed directly from a `float`.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

Nouveau dans la version 3.1.

fma (*other*, *third*, *context=None*)

Fused multiply-add. Return `self*other+third` with no rounding of the intermediate product `self*other`.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical ()

Return *True* if the argument is canonical and *False* otherwise. Currently, a *Decimal* instance is always canonical, so this operation always returns *True*.

is_finite ()

Return *True* if the argument is a finite number, and *False* if the argument is an infinity or a NaN.

is_infinite ()

Return *True* if the argument is either positive or negative infinity and *False* otherwise.

is_nan ()

Return *True* if the argument is a (quiet or signaling) NaN and *False* otherwise.

is_normal (*context=None*)

Return *True* if the argument is a *normal* finite number. Return *False* if the argument is zero, subnormal, infinite or a NaN.

is_qnan ()

Return *True* if the argument is a quiet NaN, and *False* otherwise.

is_signed ()

Return *True* if the argument has a negative sign and *False* otherwise. Note that zeros and NaNs can both carry signs.

is_snan ()

Return *True* if the argument is a signaling NaN and *False* otherwise.

is_subnormal (*context=None*)

Return *True* if the argument is subnormal, and *False* otherwise.

is_zero ()

Return *True* if the argument is a (positive or negative) zero and *False* otherwise.

ln (*context=None*)

Return the natural (base e) logarithm of the operand. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

log10 (*context=None*)

Return the base ten logarithm of the operand. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

logb (*context=None*)

For a nonzero number, return the adjusted exponent of its operand as a *Decimal* instance. If the operand is a zero then `Decimal('-Infinity')` is returned and the *DivisionByZero* flag is raised. If the operand is an infinity then `Decimal('Infinity')` is returned.

logical_and (*other*, *context=None*)

logical_and() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise and of the two operands.

logical_invert (*context=None*)

logical_invert() is a logical operation. The result is the digit-wise inversion of the operand.

logical_or (*other*, *context=None*)

logical_or() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise or of the two operands.

logical_xor (*other*, *context=None*)

logical_xor() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise exclusive or of the two operands.

max (*other*, *context=None*)

Like `max(self, other)` except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

max_mag (*other*, *context=None*)

Similar to the `max()` method, but the comparison is done using the absolute values of the operands.

min (*other*, *context=None*)

Like `min(self, other)` except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

min_mag (*other*, *context=None*)

Similar to the `min()` method, but the comparison is done using the absolute values of the operands.

next_minus (*context=None*)

Return the largest number representable in the given context (or in the current thread's context if no context is given) that is smaller than the given operand.

next_plus (*context=None*)

Return the smallest number representable in the given context (or in the current thread's context if no context is given) that is larger than the given operand.

next_toward (*other*, *context=None*)

If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

normalize (*context=None*)

Normalize the number by stripping the rightmost trailing zeros and converting any result equal to `Decimal('0')` to `Decimal('0e0')`. Used for producing canonical values for attributes of an equivalence class. For example, `Decimal('32.100')` and `Decimal('0.321000e+2')` both normalize to the equivalent value `Decimal('32.1')`.

number_class (*context=None*)

Return a string describing the *class* of the operand. The returned value is one of the following ten strings.

- `"-Infinity"`, indicating that the operand is negative infinity.
- `"-Normal"`, indicating that the operand is a negative normal number.
- `"-Subnormal"`, indicating that the operand is negative and subnormal.
- `"-Zero"`, indicating that the operand is a negative zero.
- `"+Zero"`, indicating that the operand is a positive zero.
- `"+Subnormal"`, indicating that the operand is positive and subnormal.
- `"+Normal"`, indicating that the operand is a positive normal number.
- `"+Infinity"`, indicating that the operand is positive infinity.
- `"NaN"`, indicating that the operand is a quiet NaN (Not a Number).
- `"sNaN"`, indicating that the operand is a signaling NaN.

quantize (*exp*, *rounding=None*, *context=None*)

Return a value equal to the first operand after rounding and having the exponent of the second operand.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an *InvalidOperation* is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the `rounding` argument if given, else by the given `context` argument; if neither argument is given the rounding mode of the current thread's context is used.

An error is returned whenever the resulting exponent is greater than `Emax` or less than `Etiny`.

radix ()

Return `Decimal(10)`, the radix (base) in which the *Decimal* class does all its arithmetic. Included for compatibility with the specification.

remainder_near (*other*, *context=None*)

Return the remainder from dividing *self* by *other*. This differs from `self % other` in that the sign of the

remainder is chosen so as to minimize its absolute value. More precisely, the return value is `self - n * other` where `n` is the integer nearest to the exact value of `self / other`, and if two integers are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of *self*.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate (*other*, *context=None*)

Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range `-precision` through `precision`. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length `precision` if necessary. The sign and exponent of the first operand are unchanged.

same_quantum (*other*, *context=None*)

Test whether *self* and *other* have the same exponent or whether both are NaN.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

scaleb (*other*, *context=None*)

Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by `10**other`. The second operand must be an integer.

shift (*other*, *context=None*)

Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range `-precision` through `precision`. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

sqrt (*context=None*)

Return the square root of the argument to full precision.

to_eng_string (*context=None*)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

For example, this converts `Decimal('123E+1')` to `Decimal('1.23E+3')`.

to_integral (*rounding=None*, *context=None*)

Identical to the `to_integral_value()` method. The `to_integral` name has been kept for compatibility with older versions.

to_integral_exact (*rounding=None*, *context=None*)

Round to the nearest integer, signaling *Inexact* or *Rounded* as appropriate if rounding occurs. The rounding mode is determined by the `rounding` parameter if given, else by the given `context`. If neither parameter is given then the rounding mode of the current context is used.

to_integral_value (*rounding=None*, *context=None*)

Round to the nearest integer without signaling *Inexact* or *Rounded*. If given, applies *rounding*; otherwise, uses the rounding method in either the supplied `context` or the current context.

Logical operands

The `logical_and()`, `logical_invert()`, `logical_or()`, and `logical_xor()` methods expect their arguments to be *logical operands*. A *logical operand* is a `Decimal` instance whose exponent and sign are both zero, and whose digits are all either 0 or 1.

9.4.3 Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the `getcontext()` and `setcontext()` functions :

`decimal.getcontext()`
Return the current context for the active thread.

`decimal.setcontext(c)`
Set the current context for the active thread to `c`.

You can also use the `with` statement and the `localcontext()` function to temporarily change the active context.

`decimal.localcontext(ctx=None)`
Return a context manager that will set the current context for the active thread to a copy of `ctx` on entry to the `with`-statement and restore the previous context when exiting the `with`-statement. If no context is specified, a copy of the current context is used.
For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context :

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42    # Perform a high precision calculation
    s = calculate_something()
s = +s    # Round the final result back to the default precision
```

New contexts can also be created using the `Context` constructor described below. In addition, the module provides three pre-made contexts :

class `decimal.BasicContext`

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to `ROUND_HALF_UP`. All flags are cleared. All traps are enabled (treated as exceptions) except `Inexact`, `Rounded`, and `Subnormal`.

Because many of the traps are enabled, this context is useful for debugging.

class `decimal.ExtendedContext`

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to `ROUND_HALF_EVEN`. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the traps are disabled, this context is useful for applications that prefer to have result value of `NaN` or `Infinity` instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

class `decimal.DefaultContext`

This context is used by the `Context` constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts created by the `Context` constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are `prec=28`, `rounding=ROUND_HALF_EVEN`, and enabled traps for `Overflow`, `InvalidOperation`, and `DivisionByZero`.

In addition to the three supplied contexts, new contexts can be created with the `Context` constructor.

class `decimal.Context` (`prec=None`, `rounding=None`, `Emin=None`, `Emax=None`, `capitals=None`,
`clamp=None`, `flags=None`, `traps=None`)

Creates a new context. If a field is not specified or is `None`, the default values are copied from the `DefaultContext`. If the `flags` field is not specified or is `None`, all flags are cleared.

`prec` is an integer in the range `[1, MAX_PREC]` that sets the precision for arithmetic operations in the context.

The `rounding` option is one of the constants listed in the section [Rounding Modes](#).

The `traps` and `flags` fields list any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

The `Emin` and `Emax` fields are integers specifying the outer limits allowable for exponents. `Emin` must be in the range `[MIN_EMIN, 0]`, `Emax` in the range `[0, MAX_EMAX]`.

The `capitals` field is either 0 or 1 (the default). If set to 1, exponents are printed with a capital E; otherwise, a lowercase e is used: `Decimal('6.02e+23')`.

The `clamp` field is either 0 (the default) or 1. If set to 1, the exponent `e` of a `Decimal` instance representable in this context is strictly limited to the range `Emin - prec + 1 <= e <= Emax - prec + 1`. If `clamp` is 0 then a weaker condition holds: the adjusted exponent of the `Decimal` instance is at most `Emax`. When `clamp` is 1, a large normal number will, where possible, have its exponent reduced and a corresponding number of zeros added to its coefficient, in order to fit the exponent constraints; this preserves the value of the number but loses information about significant trailing zeros. For example:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

A `clamp` value of 1 allows compatibility with the fixed-width decimal interchange formats specified in IEEE 754.

The `Context` class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the `Decimal` methods described above (with the exception of the `adjusted()` and `as_tuple()` methods) there is a corresponding `Context` method. For example, for a `Context` instance `C` and `Decimal` instance `x`, `C.exp(x)` is equivalent to `x.exp(context=C)`. Each `Context` method accepts a Python integer (an instance of `int`) anywhere that a `Decimal` instance is accepted.

clear_flags()

Resets all of the flags to 0.

clear_traps()

Resets all of the traps to 0.

Nouveau dans la version 3.3.

copy()

Return a duplicate of the context.

copy_decimal(num)

Return a copy of the `Decimal` instance `num`.

create_decimal(num)

Creates a new `Decimal` instance from `num` but using `self` as context. Unlike the `Decimal` constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace or underscores are permitted.

create_decimal_from_float(*f*)

Creates a new `Decimal` instance from a float *f* but rounding using *self* as the context. Unlike the `Decimal.from_float()` class method, the context precision, rounding method, flags, and traps are applied to the conversion.

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
...
decimal.Inexact: None
```

Nouveau dans la version 3.1.

Etiny()

Returns a value equal to $E_{\min} - \text{prec} + 1$ which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to *Etiny*.

Etop()

Returns a value equal to $E_{\max} - \text{prec} + 1$.

The usual approach to working with decimals is to create `Decimal` instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach is to use context methods for calculating within a specific context. The methods are similar to those for the `Decimal` class and are only briefly recounted here.

abs(*x*)

Renvoie la valeur absolue de *x*.

add(*x*, *y*)

Return the sum of *x* and *y*.

canonical(*x*)

Returns the same `Decimal` object *x*.

compare(*x*, *y*)

Compares *x* and *y* numerically.

compare_signal(*x*, *y*)

Compares the values of the two operands numerically.

compare_total(*x*, *y*)

Compares two operands using their abstract representation.

compare_total_mag(*x*, *y*)

Compares two operands using their abstract representation, ignoring sign.

copy_abs(*x*)

Returns a copy of *x* with the sign set to 0.

copy_negate(*x*)

Returns a copy of *x* with the sign inverted.

copy_sign(*x*, *y*)

Copies the sign from *y* to *x*.

divide(*x*, *y*)

Return *x* divided by *y*.

divide_int (*x*, *y*)
Return *x* divided by *y*, truncated to an integer.

divmod (*x*, *y*)
Divides two numbers and returns the integer part of the result.

exp (*x*)
Returns e^{**x} .

fma (*x*, *y*, *z*)
Returns *x* multiplied by *y*, plus *z*.

is_canonical (*x*)
Returns True if *x* is canonical; otherwise returns False.

is_finite (*x*)
Returns True if *x* is finite; otherwise returns False.

is_infinite (*x*)
Returns True if *x* is infinite; otherwise returns False.

is_nan (*x*)
Returns True if *x* is a qNaN or sNaN; otherwise returns False.

is_normal (*x*)
Returns True if *x* is a normal number; otherwise returns False.

is_qnan (*x*)
Returns True if *x* is a quiet NaN; otherwise returns False.

is_signed (*x*)
Returns True if *x* is negative; otherwise returns False.

is_snan (*x*)
Returns True if *x* is a signaling NaN; otherwise returns False.

is_subnormal (*x*)
Returns True if *x* is subnormal; otherwise returns False.

is_zero (*x*)
Returns True if *x* is a zero; otherwise returns False.

ln (*x*)
Returns the natural (base *e*) logarithm of *x*.

log10 (*x*)
Returns the base 10 logarithm of *x*.

logb (*x*)
Returns the exponent of the magnitude of the operand's MSD.

logical_and (*x*, *y*)
Applies the logical operation *and* between each operand's digits.

logical_invert (*x*)
Invert all the digits in *x*.

logical_or (*x*, *y*)
Applies the logical operation *or* between each operand's digits.

logical_xor (*x*, *y*)
Applies the logical operation *xor* between each operand's digits.

max (*x*, *y*)
Compares two values numerically and returns the maximum.

max_mag (*x*, *y*)
Compares the values numerically with their sign ignored.

min (*x*, *y*)
Compares two values numerically and returns the minimum.

min_mag (*x*, *y*)
Compares the values numerically with their sign ignored.

minus (*x*)

Minus corresponds to the unary prefix minus operator in Python.

multiply (*x*, *y*)

Return the product of *x* and *y*.

next_minus (*x*)

Returns the largest representable number smaller than *x*.

next_plus (*x*)

Returns the smallest representable number larger than *x*.

next_toward (*x*, *y*)

Returns the number closest to *x*, in direction towards *y*.

normalize (*x*)

Reduces *x* to its simplest form.

number_class (*x*)

Returns an indication of the class of *x*.

plus (*x*)

Plus corresponds to the unary prefix plus operator in Python. This operation applies the context precision and rounding, so it is *not* an identity operation.

power (*x*, *y*, *modulo*=None)

Return *x* to the power of *y*, reduced modulo *modulo* if given.

With two arguments, compute $x^{**}y$. If *x* is negative then *y* must be integral. The result will be inexact unless *y* is integral and the result is finite and can be expressed exactly in “precision” digits. The rounding mode of the context is used. Results are always correctly-rounded in the Python version.

Modifié dans la version 3.3 : The C module computes `power()` in terms of the correctly-rounded `exp()` and `ln()` functions. The result is well-defined but only « almost always correctly-rounded ».

With three arguments, compute $(x^{**}y) \% modulo$. For the three argument form, the following restrictions on the arguments hold :

- all three arguments must be integral
- *y* must be nonnegative
- at least one of *x* or *y* must be nonzero
- *modulo* must be nonzero and have at most “precision” digits

The value resulting from `Context.power(x, y, modulo)` is equal to the value that would be obtained by computing $(x^{**}y) \% modulo$ with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of *x*, *y* and *modulo*. The result is always exact.

quantize (*x*, *y*)

Returns a value equal to *x* (rounded), having the exponent of *y*.

radix ()

Just returns 10, as this is Decimal, :)

remainder (*x*, *y*)

Returns the remainder from integer division.

The sign of the result, if non-zero, is the same as that of the original dividend.

remainder_near (*x*, *y*)

Returns $x - y * n$, where *n* is the integer nearest the exact value of x / y (if the result is 0 then its sign will be the sign of *x*).

rotate (*x*, *y*)

Returns a rotated copy of *x*, *y* times.

same_quantum (*x*, *y*)

Returns `True` if the two operands have the same exponent.

scaleb (*x*, *y*)

Returns the first operand after adding the second value its exp.

shift (*x*, *y*)

Returns a shifted copy of *x*, *y* times.

sqrt (*x*)

Square root of a non-negative number to context precision.

subtract (*x*, *y*)Return the difference between *x* and *y*.**to_eng_string** (*x*)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

to_integral_exact (*x*)

Rounds to an integer.

to_sci_string (*x*)

Converts a number to a string using scientific notation.

9.4.4 Constantes

The constants in this section are only relevant for the C module. They are also included in the pure Python version for compatibility.

	32-bit	64-bit
<code>decimal.MAX_PREC</code>	425000000	999999999999999999
<code>decimal.MAX_EMAX</code>	425000000	999999999999999999
<code>decimal.MIN_EMIN</code>	-425000000	-999999999999999999
<code>decimal.MIN_ETINY</code>	-849999999	-1999999999999999997

`decimal.HAVE_THREADS`

The default value is `True`. If Python is compiled without threads, the C version automatically disables the expensive thread local context machinery. In this case, the value is `False`.

9.4.5 Rounding modes

`decimal.ROUND_CEILING`

Round towards Infinity.

`decimal.ROUND_DOWN`

Round towards zero.

`decimal.ROUND_FLOOR`

Round towards -Infinity.

`decimal.ROUND_HALF_DOWN`

Round to nearest with ties going towards zero.

`decimal.ROUND_HALF_EVEN`

Round to nearest with ties going to nearest even integer.

`decimal.ROUND_HALF_UP`

Round to nearest with ties going away from zero.

`decimal.ROUND_UP`

Round away from zero.

`decimal.ROUND_05UP`

Round away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise round towards zero.

9.4.6 Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the *DivisionByZero* trap is set, then a *DivisionByZero* exception is raised upon encountering the condition.

class `decimal.Clamped`

Altered an exponent to fit representation constraints.

Typically, clamping occurs when an exponent falls outside the context's `Emin` and `Emax` limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class `decimal.DecimalException`

Base class for other signals and a subclass of *ArithmeticError*.

class `decimal.DivisionByZero`

Signals the division of a non-infinite number by zero.

Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns `Infinity` or `-Infinity` with the sign determined by the inputs to the calculation.

class `decimal.Inexact`

Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

class `decimal.InvalidOperation`

An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns `NaN`. Possible causes include :

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class `decimal.Overflow`

Numerical overflow.

Indicates the exponent is larger than `Emax` after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to `Infinity`. In either case, *Inexact* and *Rounded* are also signaled.

class decimal.Rounded

Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding 5.00 to 5.0). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class decimal.Subnormal

Exponent was lower than `Emin` prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

class decimal.Underflow

Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. *Inexact* and *Subnormal* are also signaled.

class decimal.FloatOperation

Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the *Decimal* constructor, *create_decimal()* and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting *FloatOperation* in the context flags. Explicit conversions with *from_float()* or *create_decimal_from_float()* do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise *FloatOperation*.

The following table summarizes the hierarchy of signals :

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 Floating Point Notes

Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent 0.1 exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition :

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

The `decimal` module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance :

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

Special values

The number system for the `decimal` module provides special values including NaN, sNaN, -Infinity, Infinity, and two zeros, +0 and -0.

Infinities can be constructed directly with `Decimal('Infinity')`. Also, they can arise from dividing by zero when the `DivisionByZero` signal is not trapped. Likewise, when the `Overflow` signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return NaN, or if the `InvalidOperation` signal is trapped, raise an exception. For example, `0/0` returns NaN which means « not a number ». This variety of NaN is quiet and, once created, will flow through other computations always resulting in another NaN. This behavior can be useful for a series of computations that occasionally have missing inputs — it allows the calculation to proceed while flagging specific results as invalid.

A variant is sNaN which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python's comparison operators can be a little surprising where a NaN is involved. A test for equality where one of the operands is a quiet or signaling NaN always returns `False` (even when doing `Decimal('NaN')==Decimal('NaN')`), while a test for inequality always returns `True`. An attempt to compare two Decimals using any of the `<`, `<=`, `>` or `>=` operators will raise the `InvalidOperation` signal if either operand is a NaN, and return `False` if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a NaN were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the `compare()` and `compare-signal()` methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as

equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating point representations, it is not immediately obvious that the following calculation returns a value equal to zero :

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 Working with threads

The `getcontext()` function accesses a different `Context` object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext().prec=10`) without interfering with other threads.

Likewise, the `setcontext()` function automatically assigns its target to the current thread.

If `setcontext()` has not been called before `getcontext()`, then `getcontext()` will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called `DefaultContext`. To control the defaults so that each thread will use the same values throughout the application, directly modify the `DefaultContext` object. This should be done *before* any threads are started so that there won't be a race condition between threads calling `getcontext()`. For example :

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
. . .
```

9.4.9 Cas pratiques

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the `Decimal` class :

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg: optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
```

(suite sur la page suivante)

(suite de la page précédente)

```

'- $1,234,567.89'
>>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
'1.234.568-'
>>> moneyfmt(d, curr='$', neg='(', trailneg=')')
'($1,234,567.89)'
>>> moneyfmt(Decimal(123456789), sep=' ')
'123 456 789.00'
>>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
'<0.02>'

"""
q = Decimal(10) ** -places          # 2 places --> '0.01'
sign, digits, exp = value.quantize(q).as_tuple()
result = []
digits = list(map(str, digits))
build, next = result.append, digits.pop
if sign:
    build(trailneg)
for i in range(places):
    build(next() if digits else '0')
if places:
    build(dp)
if not digits:
    build('0')
i = 0
while digits:
    build(next())
    i += 1
    if i == 3 and digits:
        i = 0
    build(sep)
build(curr)
build(neg if sign else pos)
return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2 # extra digits for intermediate steps
    three = Decimal(3)     # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s                # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x. Result type matches input type.

```

(suite sur la page suivante)

(suite de la page précédente)

```

>>> print(exp(Decimal(1)))
2.718281828459045235360287471
>>> print(exp(Decimal(2)))
7.389056098930650227230427461
>>> print(exp(2.0))
7.38905609893
>>> print(exp(2+0j))
(7.38905609893+0j)

"""
getcontext().prec += 2
i, lasts, s, fact, num = 0, 0, 1, 1, 1
while s != lasts:
    lasts = s
    i += 1
    fact *= i
    num *= x
    s += num / fact
getcontext().prec -= 2
return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.4794255386042030002732879352
    >>> print(sin(0.5))
    0.479425538604

```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> print(sin(0.5+0j))
(0.479425538604+0j)

"""
getcontext().prec += 2
i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
while s != lasts:
    lasts = s
    i += 2
    fact *= i * (i-1)
    num *= x * x
    sign *= -1
    s += num / fact * sign
getcontext().prec -= 2
return +s
```

9.4.10 FAQ *decimal*

Q. C'est fastidieux de taper `decimal.Decimal('1234.5')`. Y a-t-il un moyen de réduire la frappe quand on utilise l'interpréteur interactif ?

R. Certains utilisateurs abrègent le constructeur en une seule lettre :

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used ?

A. The `quantize()` method rounds to a fixed number of decimal places. If the *Inexact* trap is set, it is also useful for validation :

```
>>> TWOPLACES = Decimal(10) ** -2           # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
...
Inexact: None
```

Q. Once I have valid two place inputs, how do I maintain that invariant throughout an application ?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a `quantize()` step :

```

>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                             # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                             # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)       # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)       # And quantize division
Decimal('0.03')

```

In developing fixed-point applications, it is convenient to define functions to handle the `quantize()` step :

```

>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)

```

```

>>> mul(a, b)                          # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')

```

Q. There are many ways to express the same value. The numbers 200, 200.000, 2E2, and 02E+4 all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value ?

A. The `normalize()` method maps all equivalent values to a single representative :

```

>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]

```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation ?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing 5.0E+3 as 5000 keeps the value constant but cannot show the original's two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeroes, losing significance, but keeping the value unchanged :

```

>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()

```

```

>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')

```

Q. Is there a way to convert a regular float to a *Decimal*?

A. Yes, any binary floating point number can be exactly expressed as a *Decimal* though an exact conversion may take more precision than intuition would suggest :

```

>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')

```

Q. Within a complex calculation, how can I make sure that I haven't gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that « what you type is what you get ». A disadvantage is that the results can look odd if you forget that the inputs haven't been rounded :

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation :

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the `Context.create_decimal()` method :

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

9.5 fractions — Nombres rationnels

Code source : [Lib/fractions.py](#)

Le module `fractions` fournit un support de l'arithmétique des nombres rationnels.

Une instance de `Fraction` peut être construite depuis une paire d'entiers, depuis un autre nombre rationnel, ou depuis une chaîne de caractères.

```
class fractions.Fraction (numerator=0, denominator=1)
class fractions.Fraction (other_fraction)
class fractions.Fraction (float)
class fractions.Fraction (decimal)
class fractions.Fraction (string)
```

La première version demande que `numerator` et `denominator` soient des instance de `numbers.Rational` et renvoie une instance de `Fraction` valant `numerator/denominator`. Si `denominator` vaut 0, une `ZeroDivisionError` est levée. La seconde version demande que `other_fraction` soit une instance de `numbers.Rational` et renvoie une instance de `Fraction` avec la même valeur. Les deux versions suivantes acceptent un `float` ou une instance de `decimal.Decimal`, et renvoient une instance de `Fraction` avec exactement la même valeur. Notez que les problèmes usuels des virgules flottantes binaires (voir `tut-fp-issues`) font que `Fraction(1.1)` n'est pas exactement égal à `11/10`, et donc `Fraction(1.1)` ne renvoie pas `Fraction(11, 10)` comme on pourrait le penser. (Mais référez-vous à la documentation de la méthode `limit_denominator()` ci-dessous.) La dernière version du constructeur attend une chaîne de caractères ou Unicode. La représentation habituelle de cette forme est :


```
[sign] numerator ['/'] denominator]
```

où le `sign` optionnel peut être soit `+` soit `-`, et `numerator` et `denominator` (si présent) sont des chaînes de chiffres décimaux. De plus, toute chaîne qui représente une valeur finie et acceptée par le constructeur de `float` est aussi acceptée par celui de `Fraction`. Dans ces deux formes, la chaîne d'entrée peut aussi contenir des espaces en début ou en fin de chaîne. Voici quelques exemples :

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

La classe `Fraction` hérite de la classe abstraite `numbers.Rational`, et implémente toutes les méthodes et opérations de cette classe. Les instances de `Fraction` sont hachables, et doivent être traitées comme immuables. En plus de cela, `Fraction` possède les propriétés et méthodes suivantes :

Modifié dans la version 3.2 : Le constructeur de `Fraction` accepte maintenant des instances de `float` et `decimal.Decimal`.

numerator

Numérateur de la fraction irréductible.

denominator

Dénominateur de la fraction irréductible.

from_float (*flt*)

Cette méthode de classe construit un objet `Fraction` représentant la valeur exacte de *flt*, qui doit être de type `float`. Attention, `Fraction.from_float(0.3)` n'est pas la même valeur que `Fraction(3, 10)`.

Note : Depuis Python 3.2, vous pouvez aussi construire une instance de `Fraction` directement depuis un `float`.

from_decimal (*dec*)

Cette méthode de classe construit un objet `Fraction` représentant la valeur exacte de *dec*, qui doit être de type `decimal.Decimal`.

Note : Depuis Python 3.2, vous pouvez aussi construire une instance de `Fraction` directement depuis une instance de `decimal.Decimal`.

limit_denominator (*max_denominator=1000000*)

Trouve et renvoie la *Fraction* la plus proche de *self* qui a au plus *max_denominator* comme dénominateur. Cette méthode est utile pour trouver des approximations rationnelles de nombres flottants donnés :

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

ou pour retrouver un nombre rationnel représenté par un flottant :

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__ ()

Renvoie le plus grand *int* \leq *self*. Cette méthode peut aussi être utilisée à travers la fonction *math.floor()*.

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__ ()

Renvoie le plus petit *int* \geq *self*. Cette méthode peut aussi être utilisée à travers la fonction *math.ceil()*.

__round__ ()

__round__ (*ndigits*)

La première version renvoie l'*int* le plus proche de *self*, arrondissant les demis au nombre pair le plus proche. La seconde version arrondit *self* au plus proche multiple de *Fraction(1, 10**ndigits)* (logiquement, si *ndigits* est négatif), arrondissant toujours les demis au nombre pair le plus proche. Cette méthode peut aussi être utilisée à via la fonction *round()*.

fractions.gcd (*a*, *b*)

Renvoie le plus grand diviseur commun (PGCD) des entiers *a* et *b*. Si *a* et *b* sont tous deux non nuls, alors la valeur absolue de *gcd(a, b)* est le plus grand entier qui divise à la fois *a* et *b*. *gcd(a, b)* a le même signe que *b* si *b* n'est pas nul ; autrement il prend le signe de *a*. *gcd(0, 0)* renvoie 0.

Obsolète depuis la version 3.5 : Utilisez plutôt *math.gcd()*.

Voir aussi :

Module *numbers* Les classes abstraites représentant la hiérarchie des nombres.

9.6 random — Génère des nombres pseudo-aléatoires

Code source : [Lib/random.py](#)

Ce module implémente des générateurs de nombres pseudo-aléatoires pour différentes distributions.

Pour les entiers, il existe une sélection uniforme à partir d'une plage. Pour les séquences, il existe une sélection uniforme d'un élément aléatoire, une fonction pour générer une permutation aléatoire d'une liste sur place et une fonction pour un échantillonnage aléatoire sans remplacement.

Pour l'ensemble des réels, il y a des fonctions pour calculer des distributions uniformes, normales (gaussiennes), log-normales, exponentielles négatives, gamma et bêta. Pour générer des distributions d'angles, la distribution de *von Mises* est disponible.

Presque toutes les fonctions du module dépendent de la fonction de base `random()`, qui génère un nombre à virgule flottante aléatoire de façon uniforme dans la plage semi-ouverte `[0.0, 1.0)`. Python utilise l'algorithme *Mersenne Twister* comme générateur de base. Il produit des flottants de précision de 53 bits et a une période de $2^{19937}-1$. L'implémentation sous-jacente en C est à la fois rapide et compatible avec les programmes ayant de multiples fils d'exécution. Le *Mersenne Twister* est l'un des générateurs de nombres aléatoires les plus largement testés qui existent. Cependant, étant complètement déterministe, il n'est pas adapté à tous les usages et est totalement inadapté à des fins cryptographiques.

Les fonctions fournies par ce module dépendent en réalité de méthodes d'une instance cachée de la classe `random.Random`. Vous pouvez créer vos propres instances de `Random` pour obtenir des générateurs sans états partagés.

La classe `Random` peut également être sous-classée si vous voulez utiliser un générateur de base différent, de votre propre conception. Dans ce cas, remplacez les méthodes `random()`, `seed()`, `getstate()` et `setstate()`. En option, un nouveau générateur peut fournir une méthode `getrandbits()` — ce qui permet à `randrange()` de produire des sélections sur une plage de taille arbitraire.

Le module `random` fournit également la classe `SystemRandom` qui utilise la fonction système `os.urandom()` pour générer des nombres aléatoires à partir de sources fournies par le système d'exploitation.

Avertissement : Les générateurs pseudo-aléatoires de ce module ne doivent pas être utilisés à des fins de sécurité. Pour des utilisations de sécurité ou cryptographiques, voir le module `secrets`.

Voir aussi :

M. Matsumoto and T. Nishimura, « Mersenne Twister : A 623-dimensionally equidistributed uniform pseudorandom number generator », ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, Janvier pp.3–30 1998.

[Complementary-Multiply-with-Carry recipe](#) pour un autre générateur de nombres aléatoires avec une longue période et des opérations de mise à jour relativement simples.

9.6.1 Fonctions de gestion d'état

`random.seed(a=None, version=2)`

Initialise le générateur de nombres aléatoires.

Si `a` est omis ou `None`, l'heure système actuelle est utilisée. Si des sources aléatoires sont fournies par le système d'exploitation, elles sont utilisées à la place de l'heure système (voir la fonction `os.urandom()` pour les détails sur la disponibilité).

Si `a` est un entier, il est utilisé directement.

Avec la version 2 (par défaut), un objet `str`, `bytes` ou `bytearray` est converti en `int` et tous ses bits sont utilisés.

Avec la version 1 (fournie pour reproduire des séquences aléatoires produites par d'anciennes versions de Python), l'algorithme pour `str` et `bytes` génère une gamme plus étroite de graines.

Modifié dans la version 3.2 : Passée à la version 2 du schéma qui utilise tous les bits d'une graine de chaîne de caractères.

`random.getstate()`

Renvoie un objet capturant l'état interne actuel du générateur. Cet objet peut être passé à `setstate()` pour restaurer cet état.

`random.setstate(state)`

Il convient que `state` ait été obtenu à partir d'un appel précédent à `getstate()`, et `setstate()` restaure l'état interne du générateur à ce qu'il était au moment où `getstate()` a été appelé.

`random.getrandbits(k)`

Renvoie un entier Python avec k bits aléatoires. Cette méthode est fournie avec le générateur MersenneTwister. Quelques autres générateurs peuvent également la fournir en option comme partie de l'API. Lorsqu'elle est disponible, `getrandbits()` permet à `randrange()` de gérer des gammes arbitrairement larges.

9.6.2 Fonctions pour les entiers

`random.randrange(stop)`

`random.randrange(start, stop[, step])`

Renvoie un élément sélectionné aléatoirement à partir de `range(start, stop, step)`. C'est équivalent à `choice(range(start, stop, step))`, mais ne construit pas réellement un objet `range`.

Le motif d'argument positionnel correspond à celui de `range()`. N'utilisez pas d'arguments nommés parce que la fonction peut les utiliser de manière inattendue.

Modifié dans la version 3.2 : `randrange()` est plus sophistiquée dans la production de valeurs uniformément distribuées. Auparavant, elle utilisait un style comme `int(random()*n)` qui pouvait produire des distributions légèrement inégales.

`random.randint(a, b)`

Renvoie un entier aléatoire N tel que $a \leq N \leq b$. Alias pour `randrange(a, b+1)`.

9.6.3 Fonctions pour les séquences

`random.choice(seq)`

Renvoie un élément aléatoire de la séquence non vide `seq`. Si `seq` est vide, lève `IndexError`.

`random.choices(population, weights=None, *, cum_weights=None, k=1)`

Renvoie une liste de taille k d'éléments choisis dans la `population` avec remise. Si la `population` est vide, lève `IndexError`.

Si une séquence de *poids* est spécifiée, les tirages sont effectués en fonction des poids relatifs. Alternativement, si une séquence `cum_weights` est donnée, les tirages sont faits en fonction des poids cumulés (peut-être calculés en utilisant `itertools.accumulate()`). Par exemple, les poids relatifs `[10, 5, 30, 5]` sont équivalents aux poids cumulatifs `[10, 15, 45, 50]`. En interne, les poids relatifs sont convertis en poids cumulatifs avant d'effectuer les tirages, ce qui vous permet d'économiser du travail en fournissant des pondérations cumulatives.

Si ni `weights` ni `cum_weights` ne sont spécifiés, les tirages sont effectués avec une probabilité uniforme. Si une séquence de poids est fournie, elle doit être de la même longueur que la séquence `population`. Spécifier à la fois `weights` et `cum_weights` lève une `TypeError`.

The `weights` or `cum_weights` can use any numeric type that interoperates with the `float` values returned by `random()` (that includes integers, floats, and fractions but excludes decimals).

Nouveau dans la version 3.6.

`random.shuffle(x[, random])`

Mélange la séquence `x` sans créer de nouvelle instance (« sur place »).

L'argument optionnel `random` est une fonction sans argument renvoyant un nombre aléatoire à virgule flottante dans `[0.0, 1.0)`; par défaut, c'est la fonction `random()`.

Pour mélanger une séquence immuable et renvoyer une nouvelle liste mélangée, utilisez `sample(x, k=len(x))` à la place.

Notez que même pour les petits `len(x)`, le nombre total de permutations de `x` peut rapidement devenir plus grand que la période de la plupart des générateurs de nombres aléatoires. Cela implique que la plupart des permutations d'une longue séquence ne peuvent jamais être générées. Par exemple, une séquence de longueur 2080 est la plus grande qui puisse tenir dans la période du générateur de nombres aléatoires Mersenne Twister.

`random.sample(population, k)`

Renvoie une liste de k éléments uniques choisis dans la séquence ou l'ensemble de la population. Utilisé pour un tirage aléatoire sans remise.

Renvoie une nouvelle liste contenant des éléments de la population tout en laissant la population originale inchangée. La liste résultante est classée par ordre de sélection de sorte que toutes les sous-tranches soient également des échantillons aléatoires valides. Cela permet aux gagnants du tirage (l'échantillon) d'être divisés en gagnants du grand prix et en gagnants de la deuxième place (les sous-tranches).

Les membres de la population n'ont pas besoin d'être *hachables* ou uniques. Si la population contient des répétitions, alors chaque occurrence est un tirage possible dans l'échantillon.

Pour choisir un échantillon parmi un intervalle d'entiers, utilisez un objet `range()` comme argument. Ceci est particulièrement rapide et économe en mémoire pour un tirage dans une grande population : `échantillon(range(10000000), k=60)`.

Si la taille de l'échantillon est supérieure à la taille de la population, une `ValueError` est levée.

9.6.4 Distributions pour les nombre réels

Les fonctions suivantes génèrent des distributions spécifiques en nombre réels. Les paramètres de fonction sont nommés d'après les variables correspondantes de l'équation de la distribution, telles qu'elles sont utilisées dans la pratique mathématique courante ; la plupart de ces équations peuvent être trouvées dans tout document traitant de statistiques.

`random.random()`

Renvoie le nombre aléatoire à virgule flottante suivant dans la plage $[0.0, 1.0)$.

`random.uniform(a, b)`

Renvoie un nombre aléatoire à virgule flottante N tel que $a \leq N \leq b$ pour $a \leq b$ et $b \leq N \leq a$ pour $b < a$.

La valeur finale b peut ou non être incluse dans la plage selon l'arrondi à virgule flottante dans l'équation $a + (b-a) * \text{random}()$.

`random.triangular(low, high, mode)`

Renvoie un nombre aléatoire en virgule flottante N tel que $\text{low} \leq N \leq \text{high}$ et avec le *mode* spécifié entre ces bornes. Les limites *low* et *high* par défaut sont zéro et un. L'argument *mode* est par défaut le point médian entre les bornes, ce qui donne une distribution symétrique.

`random.betavariate(alpha, beta)`

Distribution bêta. Les conditions sur les paramètres sont $\alpha > 0$ et $\beta > 0$. Les valeurs renvoyées varient entre 0 et 1.

`random.expovariate(lambd)`

Distribution exponentielle. *lambd* est $1,0$ divisé par la moyenne désirée. Ce ne doit pas être zéro. (Le paramètre aurait dû s'appeler « *lambda* », mais c'est un mot réservé en Python.) Les valeurs renvoyées vont de 0 à plus l'infini positif si *lambd* est positif, et de moins l'infini à 0 si *lambd* est négatif.

`random.gammavariate(alpha, beta)`

Distribution gamma. (*Ce n'est pas la fonction gamma !*) Les conditions sur les paramètres sont $\alpha > 0$ et $\beta > 0$.

La fonction de distribution de probabilité est :

$$\text{pdf}(x) = \frac{x^{(\alpha - 1)} * \text{math.exp}(-x / \beta)}{\text{math.gamma}(\alpha) * \beta^{\alpha}}$$

`random.gauss(mu, sigma)`

Distribution gaussienne. *mu* est la moyenne et *sigma* est la écart type. C'est légèrement plus rapide que la fonction `normalvariate()` définie ci-dessous.

`random.lognormvariate(mu, sigma)`

Logarithme de la distribution normale. Si vous prenez le logarithme naturel de cette distribution, vous obtiendrez une distribution normale avec *mu* moyen et écart-type *sigma*. *mu* peut avoir n'importe quelle valeur et *sigma* doit être supérieur à zéro.

`random.normalvariate(mu, sigma)`

Distribution normale. *mu* est la moyenne et *sigma* est l'écart type.

`random.vonmisesvariate(mu, kappa)`

mu est l'angle moyen, exprimé en radians entre 0 et 2π , et *kappa* est le paramètre de concentration, qui doit être supérieur ou égal à zéro. Si *kappa* est égal à zéro, cette distribution se réduit à un angle aléatoire uniforme sur la plage de 0 à 2π .

`random.paretovariate(alpha)`

Distribution de Pareto. *alpha* est le paramètre de forme.

`random.weibullvariate(alpha, beta)`

Distribution de Weibull. *alpha* est le paramètre de l'échelle et *beta* est le paramètre de forme.

9.6.5 Générateur alternatif

`class random.SystemRandom([seed])`

Classe qui utilise la fonction `os.urandom()` pour générer des nombres aléatoires à partir de sources fournies par le système d'exploitation. Non disponible sur tous les systèmes. Ne repose pas sur un état purement logiciel et les séquences ne sont pas reproductibles. Par conséquent, la méthode `seed()` n'a aucun effet et est ignorée. Les méthodes `getstate()` et `setstate()` lèvent `NotImplementedError` si vous les appelez.

9.6.6 Remarques sur la reproductibilité

Il est parfois utile de pouvoir reproduire les séquences données par un générateur de nombres pseudo-aléatoires. En réutilisant la même graine, la même séquence devrait être reproductible d'une exécution à l'autre tant que plusieurs processus ne sont pas en cours.

La plupart des algorithmes et des fonctions de génération de graine du module aléatoire sont susceptibles d'être modifiés d'une version à l'autre de Python, mais deux aspects sont garantis de ne pas changer :

- Si une nouvelle méthode de génération de graine est ajoutée, une fonction rétro-compatible sera offerte.
- La méthode `random()` du générateur continuera à produire la même séquence lorsque la fonction de génération de graine compatible recevra la même semence.

9.6.7 Exemples et recettes

Exemples de base :

```
>>> random()                                # Random float:  0.0 <= x < 1.0
0.37444488717564646

>>> uniform(2.5, 10.0)                      # Random float:  2.5 <= x < 10.0
3.1800146073117523

>>> expovariate(1 / 5)                      # Interval between arrivals averaging 5_
↪seconds
5.148957571865031

>>> randrange(10)                          # Integer from 0 to 9 inclusive
```

(suite sur la page suivante)

(suite de la page précédente)

```

7
>>> randrange(0, 101, 2)           # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw']) # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                   # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4) # Four samples without replacement
[40, 10, 50, 30]

```

Simulations :

```

>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck of 52 playing cards
>>> # and determine the proportion of cards with a ten-value
>>> # (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = sample(list(deck.elements()), k=20)
>>> seen.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> trial = lambda: choices('HT', cum_weights=(0.60, 1.00), k=7).count('H') >= 5
>>> sum(trial() for i in range(10000)) / 10000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> trial = lambda : 2500 <= sorted(choices(range(10000), k=5))[2] < 7500
>>> sum(trial() for i in range(10000)) / 10000
0.7958

```

Exemple de **bootstrapping** statistique utilisant le ré-échantillonnage avec remise pour estimer un intervalle de confiance pour la moyenne d'un échantillon de taille cinq :

```

# http://statistics.about.com/od/Applications/a/Example-Of-Bootstrapping.htm
from statistics import mean
from random import choices

data = 1, 2, 4, 4, 10
means = sorted(mean(choices(data, k=5)) for i in range(20))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[1]:.1f} to {means[-2]:.1f}')

```

Exemple d'un **resampling permutation test** pour déterminer la signification statistique ou valeur p d'une différence observée entre les effets d'un médicament et ceux d'un placebo :

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

Simulation des heures d'arrivée et des livraisons de services dans une seule file d'attente de serveurs :

```
from random import expovariate, gauss
from statistics import mean, median, stdev

average_arrival_interval = 5.6
average_service_time = 5.0
stdev_service_time = 0.5

num_waiting = 0
arrivals = []
starts = []
arrival = service_end = 0.0
for i in range(20000):
    if arrival <= service_end:
        num_waiting += 1
        arrival += expovariate(1.0 / average_arrival_interval)
        arrivals.append(arrival)
    else:
        num_waiting -= 1
        service_start = service_end if num_waiting else arrival
        service_time = gauss(average_service_time, stdev_service_time)
        service_end = service_start + service_time
        starts.append(service_start)

waits = [start - arrival for arrival, start in zip(arrivals, starts)]
print(f'Mean wait: {mean(waits):.1f}. Stdev wait: {stdev(waits):.1f}.')
print(f'Median wait: {median(waits):.1f}. Max wait: {max(waits):.1f}.')
```

Voir aussi :

[Statistics for Hackers](#) un tutoriel vidéo par [Jake Vanderplas](#) sur l'analyse statistique en utilisant seulement quelques concepts fondamentaux dont la simulation, l'échantillonnage, le brassage et la validation croisée.

[Economics Simulation](#) simulation d'un marché par [Peter Norvig](#) qui montre l'utilisation efficace de plusieurs des outils et distributions fournis par ce module (*gauss*, *uniform*, *sample*, *betavariate*, *choice*, *triangular*, et *randrange*).

[A Concrete Introduction to Probability \(using Python\)](#) un tutoriel par [Peter Norvig](#) couvrant les bases de la théorie des

probabilités, comment écrire des simulations, et comment effectuer des analyses de données avec Python.

9.7 statistics — Mathematical statistics functions

Nouveau dans la version 3.4.

Code source : [Lib/statistics.py](#)

This module provides functions for calculating mathematical statistics of numeric (Real-valued) data.

Note : Unless explicitly noted otherwise, these functions support `int`, `float`, `decimal.Decimal` and `fractions.Fraction`. Behaviour with other types (whether in the numeric tower or not) is currently unsupported. Mixed types are also undefined and implementation-dependent. If your input data consists of mixed types, you may be able to use `map()` to ensure a consistent result, e.g. `map(float, input_data)`.

9.7.1 Averages and measures of central location

These functions calculate an average or typical value from a population or sample.

<code>mean()</code>	Arithmetic mean (« average ») of data.
<code>harmonic_mean()</code>	Harmonic mean of data.
<code>median()</code>	Median (middle value) of data.
<code>median_low()</code>	Low median of data.
<code>median_high()</code>	High median of data.
<code>median_grouped()</code>	Median, or 50th percentile, of grouped data.
<code>mode()</code>	Mode (most common value) of discrete data.

9.7.2 Measures of spread

These functions calculate a measure of how much the population or sample tends to deviate from the typical or average values.

<code>pstdev()</code>	Population standard deviation of data.
<code>pvariance()</code>	Population variance of data.
<code>stdev()</code>	Sample standard deviation of data.
<code>variance()</code>	Sample variance of data.

9.7.3 Function details

Note : The functions do not require the data given to them to be sorted. However, for reading convenience, most of the examples show sorted sequences.

`statistics.mean(data)`

Return the sample arithmetic mean of *data* which can be a sequence or iterator.

The arithmetic mean is the sum of the data divided by the number of data points. It is commonly called « the average », although it is only one of many different mathematical averages. It is a measure of the central location of the data.

If *data* is empty, `StatisticsError` will be raised.

Some examples of use :

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

Note : The mean is strongly affected by outliers and is not a robust estimator for central location : the mean is not necessarily a typical example of the data points. For more robust, although less efficient, measures of central location, see `median()` and `mode()`. (In this case, « efficient » refers to statistical efficiency rather than computational efficiency.)

The sample mean gives an unbiased estimate of the true population mean, which means that, taken on average over all the possible samples, `mean(sample)` converges on the true mean of the entire population. If *data* represents the entire population rather than a sample, then `mean(data)` is equivalent to calculating the true population mean μ .

`statistics.harmonic_mean(data)`

Return the harmonic mean of *data*, a sequence or iterator of real-valued numbers.

The harmonic mean, sometimes called the subcontrary mean, is the reciprocal of the arithmetic `mean()` of the reciprocals of the data. For example, the harmonic mean of three values *a*, *b* and *c* will be equivalent to $3 / (1/a + 1/b + 1/c)$.

The harmonic mean is a type of average, a measure of the central location of the data. It is often appropriate when averaging quantities which are rates or ratios, for example speeds. For example :

Suppose an investor purchases an equal value of shares in each of three companies, with P/E (price/earning) ratios of 2.5, 3 and 10. What is the average P/E ratio for the investor's portfolio?

```
>>> harmonic_mean([2.5, 3, 10]) # For an equal investment portfolio.
3.6
```

Using the arithmetic mean would give an average of about 5.167, which is too high.

`StatisticsError` is raised if *data* is empty, or any element is less than zero.

Nouveau dans la version 3.6.

`statistics.median(data)`

Return the median (middle value) of numeric data, using the common « mean of middle two » method. If *data* is empty, `StatisticsError` is raised. *data* can be a sequence or iterator.

The median is a robust measure of central location, and is less affected by the presence of outliers in your data. When the number of data points is odd, the middle data point is returned :

```
>>> median([1, 3, 5])
3
```

When the number of data points is even, the median is interpolated by taking the average of the two middle values :

```
>>> median([1, 3, 5, 7])
4.0
```

This is suited for when your data is discrete, and you don't mind that the median may not be an actual data point. If your data is ordinal (supports order operations) but not numeric (doesn't support addition), you should use `median_low()` or `median_high()` instead.

Voir aussi :

`median_low()`, `median_high()`, `median_grouped()`

`statistics.median_low(data)`

Return the low median of numeric data. If *data* is empty, `StatisticsError` is raised. *data* can be a sequence or iterator.

The low median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the smaller of the two middle values is returned.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

Use the low median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

`statistics.median_high(data)`

Return the high median of data. If *data* is empty, `StatisticsError` is raised. *data* can be a sequence or iterator.

The high median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the larger of the two middle values is returned.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

Use the high median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

`statistics.median_grouped(data, interval=1)`

Return the median of grouped continuous data, calculated as the 50th percentile, using interpolation. If *data* is empty, `StatisticsError` is raised. *data* can be a sequence or iterator.

```
>>> median_grouped([52, 52, 53, 54])
52.5
```

In the following example, the data are rounded, so that each value represents the midpoint of data classes, e.g. 1 is the midpoint of the class 0.5–1.5, 2 is the midpoint of 1.5–2.5, 3 is the midpoint of 2.5–3.5, etc. With the data given, the middle value falls somewhere in the class 3.5–4.5, and interpolation is used to estimate it :

```
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 5])
3.7
```

Optional argument *interval* represents the class interval, and defaults to 1. Changing the class interval naturally will change the interpolation :

```
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

This function does not check whether the data points are at least *interval* apart.

CPython implementation detail : Under some circumstances, *median_grouped()* may coerce data points to floats. This behaviour is likely to change in the future.

Voir aussi :

- « Statistics for the Behavioral Sciences », Frederick J Gravetter and Larry B Wallnau (8th Edition).
- Calculating the [median](#).
- The [SSMEDIAN](#) function in the Gnome Gnumeric spreadsheet, including [this discussion](#).

`statistics.mode(data)`

Return the most common data point from discrete or nominal *data*. The mode (when it exists) is the most typical value, and is a robust measure of central location.

If *data* is empty, or if there is not exactly one most common value, *StatisticsError* is raised.

mode assumes discrete data, and returns a single value. This is the standard treatment of the mode as commonly taught in schools :

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

The mode is unique in that it is the only statistic which also applies to nominal (non-numeric) data :

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

`statistics.pstdev(data, mu=None)`

Return the population standard deviation (the square root of the population variance). See *pvariance()* for arguments and other details.

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

`statistics.pvariance(data, mu=None)`

Return the population variance of *data*, a non-empty iterable of real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *mu* is given, it should be the mean of *data*. If it is missing or *None* (the default), the mean is automatically calculated.

Use this function to calculate the variance from the entire population. To estimate the variance from a sample, the *variance()* function is usually a better choice.

Raises *StatisticsError* if *data* is empty.

Examples :

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

If you have already calculated the mean of your data, you can pass it as the optional second argument *mu* to avoid recalculation :

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

This function does not attempt to verify that you have passed the actual mean as *mu*. Using arbitrary values for *mu* may lead to invalid or impossible results.

Decimals and Fractions are supported :

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

Note : When called with the entire population, this gives the population variance σ^2 . When called on a sample instead, this is the biased sample variance s^2 , also known as variance with N degrees of freedom.

If you somehow know the true population mean μ , you may use this function to calculate the variance of a sample, giving the known population mean as the second argument. Provided the data points are representative (e.g. independent and identically distributed), the result will be an unbiased estimate of the population variance.

`statistics.stdev` (*data*, *xbar=None*)

Return the sample standard deviation (the square root of the sample variance). See [variance\(\)](#) for arguments and other details.

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

`statistics.variance` (*data*, *xbar=None*)

Return the sample variance of *data*, an iterable of at least two real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *xbar* is given, it should be the mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

Use this function when your data is a sample from a population. To calculate the variance from the entire population, see [pvariance\(\)](#).

Raises `StatisticsError` if *data* has fewer than two values.

Examples :

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

If you have already calculated the mean of your data, you can pass it as the optional second argument *xbar* to avoid recalculation :

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

This function does not attempt to verify that you have passed the actual mean as *xbar*. Using arbitrary values for *xbar* can lead to invalid or impossible results.

Decimal and Fraction values are supported :

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

Note : This is the sample variance s^2 with Bessel's correction, also known as variance with N-1 degrees of freedom. Provided that the data points are representative (e.g. independent and identically distributed), the result should be an unbiased estimate of the true population variance.

If you somehow know the actual population mean μ you should pass it to the `pvariance()` function as the *mu* parameter to get the variance of a sample.

9.7.4 Exceptions

A single exception is defined :

exception `statistics.StatisticsError`

Subclass of `ValueError` for statistics-related exceptions.

Modules de programmation fonctionnelle

Les modules décrits dans ce chapitre fournissent des fonctions et des classes permettant d'adopter un style fonctionnel, ainsi que des manipulations sur des appelables.

Les modules suivants sont documentés dans ce chapitre :

10.1 `itertools` — Fonctions créant des itérateurs pour boucler efficacement

Ce module implémente de nombreuses briques *d'itérateurs* inspirées par des éléments de APL, Haskell et SML. Toutes ont été retravaillées dans un format adapté à Python.

Ce module standardise un ensemble de base d'outils rapides et efficaces en mémoire qui peuvent être utilisés individuellement ou en les combinant. Ensemble, ils forment une « algèbre d'itérateurs » rendant possible la construction rapide et efficace d'outils spécialisés en Python.

Par exemple, SML fournit un outil de tabulation `tabulate(f)` qui produit une séquence `f(0), f(1), ...`. Le même résultat peut être obtenu en Python en combinant `map()` et `count()` pour former `map(f, count())`.

Ces outils et leurs équivalents natifs fonctionnent également bien avec les fonctions optimisées du module `operator`. Par exemple, l'opérateur de multiplication peut être appliqué à deux vecteurs pour créer un produit scalaire efficace : `sum(map(operator.mul, vecteur1, vecteur2))`.

Itérateurs infinis :

Itérateur	Argu-ments	Résultats	Exemple
<code>count()</code>	start, [step]	start, start+step, start+2*step, ...	<code>count(10)</code> --> 10 11 12 13 14 ...
<code>cycle()</code>	p	p0, p1, ... plast, p0, p1, ...	<code>cycle('ABCD')</code> --> A B C D A B C D ...
<code>repeat()</code>	elem [,n]	<i>elem, elem, elem, ...</i> à l'infini ou jusqu'à n fois	<code>repeat(10, 3)</code> --> 10 10 10

Itérateurs se terminant par la séquence d'entrée la plus courte :

Itérateur	Arguments	Résultats	Exemple
<code>accumulate()</code>	p [,func]	p0, p0+p1, p0+p1+p2, ...	<code>accumulate([1,2,3,4,5])</code> --> 1 3 6 10 15
<code>chain()</code>	p, q, ...	p0, p1, ... plast, q0, q1, ...	<code>chain('ABC', 'DEF')</code> --> A B C D E F
<code>chain.from_iterable()</code>	itérable	p0, p1, ... plast, q0, q1, ...	<code>chain.from_iterable(['ABC', 'DEF'])</code> --> A B C D E F
<code>compress()</code>	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	<code>compress('ABCDEF', [1,0,1,0,1,1])</code> --> A C E F
<code>dropwhile()</code>	pred, seq	seq[n], seq[n+1], commençant quand <i>pred</i> échoue	<code>dropwhile(lambda x: x<5, [1,4,6,4,1])</code> --> 6 4 1
<code>filterfalse()</code>	pred, seq	éléments de <i>seq</i> pour lesquels <i>pred(elem)</i> est faux	<code>filterfalse(lambda x: x%2, range(10))</code> --> 0 2 4 6 8
<code>groupby()</code>	iterable[, key]	sous-itérateurs groupés par la valeur de <i>key(v)</i>	
<code>islice()</code>	seq, [start,] stop [, step]	éléments de <i>seq[start:stop:step]</i>	<code>islice('ABCDEFGH', 2, None)</code> --> C D E F G
<code>starmap()</code>	func, seq	func(*seq[0]), func(*seq[1]), ...	<code>starmap(pow, [(2,5), (3,2), (10,3)])</code> --> 32 9 1000
<code>takewhile()</code>	pred, seq	seq[0], seq[1], jusqu'à ce que <i>pred</i> échoue	<code>takewhile(lambda x: x<5, [1,4,6,4,1])</code> --> 1 4
<code>tee()</code>	it, n	<i>it1, it2, ... itm</i> sépare un itérateur en <i>n</i>	
<code>zip_longest()</code>	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	<code>zip_longest('ABCD', 'xy', fillvalue='-')</code> --> Ax By C- D-

Itérateurs combinatoires :

Itérateur	Arguments	Résultats
<code>product()</code>	<code>p, q, ... [repeat=1]</code>	produit cartésien, équivalent à une boucle <i>for</i> imbriquée
<code>permutations()</code>	<code>p[, r]</code>	n-uplets de longueur <code>r</code> , tous les ré-arrangements possibles, sans répétition d'éléments
<code>combinations()</code>	<code>p, r</code>	n-uplets de longueur <code>r</code> , ordonnés, sans répétition d'éléments
<code>combinations_with_replacement(p, r)</code>	<code>p, r</code>	n-uplets de longueur <code>r</code> , ordonnés, avec répétition d'éléments
<code>product('ABCD', repeat=2)</code>		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
<code>permutations('ABCD', 2)</code>		AB AC AD BA BC BD CA CB CD DA DB DC
<code>combinations('ABCD', 2)</code>		AB AC AD BC BD CD
<code>combinations_with_replacement('ABCD', 2)</code>		AA AB AC AD BB BC BD CC CD DD

10.1.1 Fonctions d'*itertools*

Toutes les fonctions du module qui suivent construisent et renvoient des itérateurs. Certaines produisent des flux de longueur infinie ; celles-ci ne doivent donc être contrôlées que par des fonctions ou boucles qui interrompent le flux.

`itertools.accumulate(iterable[, func])`

Créer un itérateur qui renvoie les sommes accumulées, ou les résultats accumulés d'autres fonctions binaires (spécifiées par l'argument optionnel *func*). Si *func* est renseigné, il doit être une fonction à deux arguments. Les éléments de l'itérable d'entrée peuvent être de n'importe quel type qui peuvent être acceptés comme arguments de *func*. (Par exemple, avec l'opération par défaut d'addition, les éléments peuvent être de n'importe quel type additionnable incluant *Decimal* ou *Fraction*.) Si l'itérable d'entrée est vide, l'itérable de sortie sera aussi vide.

À peu près équivalent à :

```
def accumulate(iterable, func=operator.add):
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
    it = iter(iterable)
    try:
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

Il y a de nombreuses utilisations à l'argument *func*. Celui-ci peut être `min()` pour calculer un minimum glissant, `max()` pour un maximum glissant ou `operator.mul()` pour un produit glissant. Des tableaux de remboursements peuvent être construits en ajoutant les intérêts et en soustrayant les paiements. Des suites par récurrence de premier ordre peuvent être modélisées en en passant la valeur initiale dans *iterable* et en n'utilisant que le premier argument de *func*, qui contient le résultat des évaluations précédentes :

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))      # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))              # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Amortize a 5% loan of 1000 with 4 annual payments of 90
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

# Chaotic recurrence relation https://en.wikipedia.org/wiki/Logistic_map
>>> logistic_map = lambda x, _: r * x * (1 - x)
>>> r = 3.8
>>> x0 = 0.4
>>> inputs = repeat(x0, 36) # only the initial value is used
>>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
 '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
 '0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',
 '0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']
```

Voir `functools.reduce()` pour une fonction similaire qui ne renvoie que la valeur accumulée finale.

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : Ajout du paramètre optionnel *func*.

`itertools.chain(*iterables)`

Crée un itérateur qui renvoie les éléments du premier itérable jusqu'à son épuisement, puis continue avec l'itérable suivant jusqu'à ce que tous les itérables soient épuisés. Utilisée pour traiter des séquences consécutives comme une seule séquence. À peu près équivalent à :

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

`classmethod chain.from_iterable(iterable)`

Constructeur alternatif pour `chain()`. Récupère des entrées chaînées depuis un unique itérable passé en argument, qui est évalué de manière paresseuse. À peu près équivalent à :

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

`itertools.combinations(iterable, r)`

Renvoie les combinaisons de longueur *r* de *iterable*.

Les combinaisons sont produites dans l'ordre lexicographique. Ainsi, si l'itérable *iterable* est ordonné, les n-uplets de combinaison produits le sont aussi.

Les éléments sont considérés comme uniques en fonction de leur position, et non pas de leur valeur. Ainsi, si les éléments en entrée sont uniques, il n'y aura pas de valeurs répétées dans chaque combinaison.

À peu près équivalent à :

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
```

(suite sur la page suivante)

(suite de la page précédente)

```

if r > n:
    return
indices = list(range(r))
yield tuple(pool[i] for i in indices)
while True:
    for i in reversed(range(r)):
        if indices[i] != i + n - r:
            break
    else:
        return
    indices[i] += 1
    for j in range(i+1, r):
        indices[j] = indices[j-1] + 1
    yield tuple(pool[i] for i in indices)

```

Un appel à `combinations()` peut aussi être vu comme à un appel à `permutations()` en excluant les sorties dans lesquelles les éléments ne sont pas ordonnés (avec la même relation d'ordre que pour l'entrée) :

```

def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)

```

Le nombre d'éléments renvoyés est $n! / r! / (n-r)!$ quand $0 \leq r \leq n$ ou zéro quand $r > n$.

`itertools.combinations_with_replacement(iterable, r)`

Renvoyer les sous-séquences de longueur r des éléments de l'itérable *iterable* d'entrée, permettant aux éléments individuels d'être répétés plus d'une fois.

Les combinaisons sont produites dans l'ordre lexicographique. Ainsi, si l'itérable *iterable* est ordonné, les n-uplets de combinaison produits le sont aussi.

Les éléments sont considérés comme uniques en fonction de leur position, et non pas de leur valeur. Ainsi si les éléments d'entrée sont uniques, les combinaisons générées seront aussi uniques.

À peu près équivalent à :

```

def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)

```

Un appel à `combinations_with_replacement()` peut aussi être vu comme un appel à `product()` en excluant les sorties dans lesquelles les éléments ne sont pas dans l'ordre (avec la même relation d'ordre que pour l'entrée) :

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

Le nombre d'éléments renvoyés est $(n+r-1)! / r! / (n-1)!$ quand $n > 0$.

Nouveau dans la version 3.1.

`itertools.compress(data, selectors)`

Crée un itérateur qui filtre les éléments de *data*, en ne renvoyant que ceux dont l'élément correspondant dans *selectors* s'évalue à True. S'arrête quand l'itérable *data* ou *selectors* a été épuisé. À peu près équivalent à :

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in zip(data, selectors) if s)
```

Nouveau dans la version 3.1.

`itertools.count(start=0, step=1)`

Crée un itérateur qui renvoie des valeurs espacées régulièrement, en commençant par le nombre *start*. Souvent utilisé comme un argument de `map()` pour générer des points de données consécutifs. Aussi utilisé avec `zip()` pour ajouter des nombres de séquence. À peu près équivalent à :

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

Pour compter avec des nombres à virgule flottante, il est parfois préférable d'utiliser le code : `(start + step * i for i in count())` pour obtenir une meilleure précision.

Modifié dans la version 3.1 : Ajout de l'argument *step* et ajout du support pour les arguments non-entiers.

`itertools.cycle(iterable)`

Crée un itérateur qui renvoie les éléments de l'itérable en en sauvegardant une copie. Quand l'itérable est épuisé, renvoie les éléments depuis la sauvegarde. Répète à l'infini. À peu près équivalent à :

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
            yield element
```

Note, cette fonction peut avoir besoin d'un stockage auxiliaire important (en fonction de la longueur de l'itérable).

`itertools.dropwhile(predicate, iterable)`

Crée un itérateur qui saute les éléments de l'itérable tant que le prédicat est vrai ; renvoie ensuite chaque élément. Notez que l'itérateur ne produit *aucune* sortie avant que le prédicat ne devienne faux, il peut donc avoir un temps de démarrage long. À peu près équivalent à :

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

`itertools.filterfalse` (*predicate, iterable*)

Crée un itérateur qui filtre les éléments de *iterable*, ne renvoyant seulement ceux pour lesquels le prédicat est `False`. Si *predicate* vaut `None`, renvoie les éléments qui sont faux. À peu près équivalent à :

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

`itertools.groupby` (*iterable, key=None*)

Crée un itérateur qui renvoie les clés et les groupes de l'itérable *iterable*. La clé *key* est une fonction qui génère une clé pour chaque élément. Si *key* n'est pas spécifiée ou est `None`, elle vaut par défaut une fonction d'identité qui renvoie l'élément sans le modifier. Généralement, l'itérable a besoin d'avoir ses éléments déjà classés selon cette même fonction de clé.

L'opération de `groupby()` est similaire au filtre `uniq` dans Unix. Elle génère un nouveau groupe à chaque fois que la valeur de la fonction *key* change (ce pourquoi il est souvent nécessaire d'avoir trié les données selon la même fonction de clé). Ce comportement est différent de celui de `GROUP BY` de SQL qui agrège les éléments sans prendre compte de leur ordre d'entrée.

Le groupe renvoyé est lui-même un itérateur qui partage l'itérable sous-jacent avec `groupby()`. Puisque que la source est partagée, quand l'objet `groupby()` est avancé, le groupe précédent n'est plus visible. Ainsi, si cette donnée doit être utilisée plus tard, elle doit être stockée comme une liste :

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

`groupby()` est à peu près équivalente à :

```
class groupby:
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def __next__(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```

while self.currkey == self.tgtkey:
    self.currvalue = next(self.it)    # Exit on StopIteration
    self.currkey = self.keyfunc(self.currvalue)
    self.tgtkey = self.currkey
return (self.currkey, self._grouper(self.tgtkey))
def _grouper(self, tgtkey):
    while self.currkey == tgtkey:
        yield self.currvalue
        try:
            self.currvalue = next(self.it)
        except StopIteration:
            return
    self.currkey = self.keyfunc(self.currvalue)

```

`itertools.islice(iterable, stop)`

`itertools.islice(iterable, start, stop[, step])`

Crée un itérateur qui renvoie les éléments sélectionnés de l'itérable. Si *start* est différent de zéro, alors les éléments de l'itérable sont ignorés jusqu'à ce que *start* soit atteint. Ensuite, les éléments sont renvoyés consécutivement sauf si *step* est plus grand que 1, auquel cas certains éléments seront ignorés. Si *stop* est `None`, alors l'itération continue jusqu'à ce que l'itérateur soit épuisé s'il ne l'est pas déjà; sinon, il s'arrête à la position spécifiée. À la différence des tranches standards, `slice()` ne gère pas les valeurs négatives pour *start*, *stop* ou *step*. Peut être utilisée pour extraire les champs consécutifs depuis des données dont la structure interne a été aplatie (par exemple, un rapport multi-lignes pourrait lister un nom de champ toutes les trois lignes). À peu près similaire à :

```

def islice(iterable, *args):
    # islice('ABCDEFGH', 2) --> A B
    # islice('ABCDEFGH', 2, 4) --> C D
    # islice('ABCDEFGH', 2, None) --> C D E F G
    # islice('ABCDEFGH', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass

```

Si *start* vaut `None`, alors l'itération commence à zéro. Si *step* vaut `None`, alors le pas est à 1 par défaut.

`itertools.permutations(iterable, r=None)`

Renvoie les arrangements successifs de longueur *r* des éléments de *iterable*.

Si *r* n'est pas spécifié ou vaut `None`, alors *r* a pour valeur la longueur de *iterable* et toutes les permutations de longueur *r* possibles sont générées.

Les permutations sont émises dans l'ordre lexicographique. Ainsi, si l'itérable d'entrée *iterable* est classé, les n-uplets de permutation sont produits dans ce même ordre.

Les éléments sont considérés comme uniques en fonction de leur position, et non pas de leur valeur. Ainsi, si l'élément est unique, il n'y aura pas de valeurs répétées dans chaque permutation.

À peu près équivalent à :

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

Un appel à `permutations()` peut aussi être vu un appel à `product()` en excluant les sorties avec des doublons (avec la même relation d'ordre que pour l'entrée) :

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

Le nombre d'éléments renvoyés est $n! / (n-r)!$ quand $0 \leq r \leq n$ ou zéro quand $r > n$.

`itertools.product(*iterables, repeat=1)`

Produit cartésien des itérables d'entrée.

À peu près équivalent à des boucles *for* imbriquées dans une expression de générateur. Par exemple `product(A, B)` renvoie la même chose que `((x, y) for x in A for y in B)`.

Les boucles imbriquées tournent comme un compteur kilométrique avec l'élément le plus à droite avançant à chaque itération. Ce motif définit un ordre lexicographique afin que, si les éléments des itérables en l'entrée sont ordonnés, les n-uplets produits le sont aussi.

Pour générer le produit d'un itérable avec lui-même, spécifiez le nombre de répétitions avec le paramètre nommé optionnel *repeat*. Par exemple, `product(A, repeat=4)` est équivalent à `product(A, A, A, A)`.

Cette fonction est à peu près équivalente au code suivant, à la différence près que la vraie implémentation ne crée pas de résultats intermédiaires en mémoire :

```
def product(*args, repeat=1):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
```

(suite sur la page suivante)

(suite de la page précédente)

```

pools = [tuple(pool) for pool in args] * repeat
result = [[]]
for pool in pools:
    result = [x+[y] for x in result for y in pool]
for prod in result:
    yield tuple(prod)

```

`itertools.repeat(object[, times])`

Crée un itérateur qui renvoie *object* à l'infini. S'exécute indéfiniment sauf si l'argument *times* est spécifié. Utilisée comme argument de `map()` pour les paramètres invariants de la fonction appelée. Aussi utilisée avec `zip()` pour créer une partie invariante d'un n-uplet.

À peu près équivalent à :

```

def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object

```

Une utilisation courante de *repeat* est de fournir un flux constant de valeurs à *map* ou *zip* :

```

>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

```

`itertools.starmap(function, iterable)`

Crée un itérateur qui exécute la fonction avec les arguments obtenus depuis l'itérable. Utilisée à la place de `map()` quand les arguments sont déjà groupés en n-uplets depuis un seul itérable — la donnée a déjà été « pré-zippée ». La différence entre `map()` et `starmap()` est similaire à la différence entre fonction(*a*,*b*) et fonction(**c*). À peu près équivalent à :

```

def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)

```

`itertools.takewhile(predicate, iterable)`

Crée un itérateur qui renvoie les éléments d'un itérable tant que le prédicat est vrai. À peu près équivalent à :

```

def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break

```

`itertools.tee(iterable, n=2)`

Renvoie *n* itérateurs indépendants depuis un unique itérable.

Le code Python qui suit aide à expliquer ce que fait *tee*, bien que la vraie implémentation soit plus complexe et n'utilise qu'une file FIFO.

À peu près équivalent à :


```
def tee(iterable, n=2):
    it = iter(iterable)
    deque = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:
                # when the local deque is empty
                try:
                    newval = next(it) # fetch a new value and
                except StopIteration:
                    return
            for d in deque:
                # load it to all the deque
                d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deque)
```

Une fois que `tee()` a créé un branchement, l'itérable *iterable* ne doit être utilisé nulle part ailleurs ; sinon, *iterable* pourrait être avancé sans que les objets *tee* ne soient informés.

Cet outil peut avoir besoin d'un stockage auxiliaire important (en fonction de la taille des données temporaires nécessaires). En général, si un itérateur utilise la majorité ou toute la donnée avant qu'un autre itérateur ne commence, il est plus rapide d'utiliser `list()` à la place de `tee()`.

`itertools.zip_longest(*iterables, fillvalue=None)`

Crée un itérateur qui agrège les éléments de chacun des itérables. Si les itérables sont de longueurs différentes, les valeurs manquantes sont remplacées par *fillvalue*. L'itération continue jusqu'à ce que l'itérable le plus long soit épuisé. À peu près équivalent à :

```
class ZipExhausted(Exception):
    pass

def zip_longest(*args, **kwargs):
    # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    fillvalue = kwargs.get('fillvalue')
    counter = len(args) - 1
    def sentinel():
        nonlocal counter
        if not counter:
            raise ZipExhausted
        counter -= 1
        yield fillvalue
    fillers = repeat(fillvalue)
    iterators = [chain(it, sentinel(), fillers) for it in args]
    try:
        while iterators:
            yield tuple(map(next, iterators))
    except ZipExhausted:
        pass
```

Si un des itérables est potentiellement infini, alors la fonction `zip_longest()` doit être encapsulée dans un code qui limite le nombre d'appels (par exemple, `islice()` ou `takewhile()`). Si *fillvalue* n'est pas spécifié, il vaut `None` par défaut.

10.1.2 Recettes *itertools*

Cette section présente des recettes pour créer une vaste boîte à outils en se servant des *itertools* existants comme des briques.

Ces outils dérivés offrent la même bonne performance que les outils sous-jacents. La performance mémoire supérieure est gardée en traitant les éléments un à la fois plutôt que de charger tout l'itérable en mémoire en même temps. Le volume de code reste bas grâce à un chaînage de style fonctionnel qui aide à éliminer les variables temporaires. La grande vitesse est gardée en préférant les briques « vectorisées » plutôt que les boucles *for* et les *générateurs* qui engendrent un surcoût de traitement.

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def prepend(value, iterator):
    "Prepend a single value in front of an iterator"
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([value], iterator)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def tail(n, iterable):
    "Return an iterator over the last n items"
    # tail(3, 'ABCDEFG') --> E F G
    return iter(collections.deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)

def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(map(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))
```

(suite sur la page suivante)

(suite de la page précédente)

```

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(map(operator.mul, vec1, vec2))

def flatten(listOfLists):
    "Flatten one level of nesting"
    return chain.from_iterable(listOfLists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    num_active = len(iterables)
    nexts = cycle(iter(it).__next__ for it in iterables)
    while num_active:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            # Remove the iterator we just exhausted from the cycle.
            num_active -= 1
            nexts = cycle(islice(nexts, num_active))

def partition(pred, iterable):
    "Use a predicate to partition entries into false entries and true entries"
    # partition(is_odd, range(10)) --> 0 2 4 6 8 and 1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

```

(suite sur la page suivante)

(suite de la page précédente)

```

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return map(next, map(itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like builtins.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
    iter_except(functools.partial(heappop, h), IndexError)   # priority queue_
↪ iterator
    iter_except(d.popitem, KeyError)                         # non-blocking dict_
↪ iterator
    iter_except(d.popleft, IndexError)                       # non-blocking deque_
↪ iterator
    iter_except(q.get_nowait, Queue.Empty)                  # loop over a_
↪ producer Queue
    iter_except(s.pop, KeyError)                             # non-blocking set_
↪ iterator

    """
    try:
        if first is not None:
            yield first()                # For database APIs needing an initial cast to_
↪ db.first()
        while True:
            yield func()
    except exception:
        pass

def first_true(iterable, default=False, pred=None):
    """Returns the first true value in the iterable.

    If no true value is found, returns *default*

```

(suite sur la page suivante)

(suite de la page précédente)

```

    If *pred* is not None, returns the first item
    for which pred(item) is true.

    """
    # first_true([a,b,c], x) --> a or b or c or x
    # first_true([a,b], x, f) --> a if f(a) else b if f(b) else x
    return next(filter(pred, iterable), default)

def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwargs)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in range(r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    'Equivalent to list(combinations(iterable, r))[index]'
    pool = tuple(iterable)
    n = len(pool)
    if r < 0 or r > n:
        raise ValueError
    c = 1
    k = min(r, n-r)
    for i in range(1, k+1):
        c = c * (n - k + i) // i
    if index < 0:
        index += c
    if index < 0 or index >= c:
        raise IndexError
    result = []
    while r:
        c, n, r = c*r//n, n-1, r-1
        while index >= c:
            index -= c
            c, n = c*(n-r)//n, n-1
        result.append(pool[-1-n])
    return tuple(result)

```

Note, beaucoup des recettes ci-dessus peuvent être optimisées en remplaçant les recherches globales par des recherches locales avec des variables locales définies comme des valeurs par défaut. Par exemple, la recette *dotproduct* peut être écrite comme :

```
def dotproduct(vec1, vec2, sum=sum, map=map, mul=operator.mul):
    return sum(map(mul, vec1, vec2))
```

10.2 `functools` — Fonctions de haut niveau et opérations sur des objets appelables

Code source : [Lib/functools.py](#)

Le module *functools* est utilisé pour des fonctions de haut niveau : des fonctions qui agissent sur ou renvoient d'autres fonctions. En général, tout objet callable peut être utilisé comme une fonction pour les besoins de ce module.

Le module *functools* définit les fonctions suivantes :

`functools.cmp_to_key(func)`

Transforme une fonction de comparaison à l'ancienne en une *fonction clé*. Utilisé avec des outils qui acceptent des fonctions clef (comme `sorted()`, `min()`, `max()`, `heapq.nlargest()`, `heapq.nsmallest()`, `itertools.groupby()`). Cette fonction est destinée au portage de fonctions python 2 utilisant des fonctions de comparaison vers Python 3.

Une fonction de comparaison est un callable qui prend deux arguments, les compare, et renvoie un nombre négatif pour l'infériorité, zéro pour l'égalité ou un nombre positif pour la supériorité. Une fonction de clé est un callable qui prend un argument et retourne une autre valeur qui sera utilisée comme clé de tri.

Exemple :

```
sorted(iterable, key=cmp_to_key(locale.strcoll)) # locale-aware sort order
```

Pour des exemples de tris et un bref tutoriel, consultez [sortinghowto](#).

Nouveau dans la version 3.2.

`@functools.lru_cache(maxsize=128, typed=False)`

Décorateur qui englobe une fonction avec un callable mémoisant qui enregistre jusqu'à *maxsize* appels récents. Cela peut gagner du temps quand une fonction coûteuse en ressources est souvent appelée avec les mêmes arguments.

Comme un dictionnaire est utilisé pour mettre en cache les résultats, les arguments positionnels et nommés de la fonction doivent être hachables.

Si *maxsize* est à `None`, la fonctionnalité LRU est désactivée et le cache peut grossir sans limite. La fonctionnalité LRU fonctionne mieux quand *maxsize* est une puissance de deux.

Si *typed* est vrai, les arguments de différents types seront mis en cache séparément. Par exemple, `f(3)` et `f(3.0)` seront considérés comme des appels distincts avec des résultats distincts.

Pour aider à mesurer l'efficacité du cache et ajuster le paramètre *maxsize*, la fonction englobée est surveillée avec une fonction `cache_info()` qui renvoie un *named tuple* affichant les *hits*, *misses*, *maxsize* et *currsize*. Dans un environnement *multithread*, les succès et échecs d'appel du cache sont approximatifs.

Le décorateur fournit également une fonction `cache_clear()` pour vider ou invalider le cache.

La fonction sous-jacente originale est accessible à travers l'attribut `__wrapped__`. Ceci est utile pour l'introspection, pour outrepasser le cache, ou pour ré-englober la fonction avec un cache différent.

Un *cache LRU* (**least recently used**) fonctionne très bien lorsque les appels récents sont les prochains appels les plus probables (par exemple, les articles les plus lus d'un serveur d'actualités ont tendance à ne changer que d'un jour à l'autre). La taille limite du cache permet de s'assurer que le cache ne grossisse pas sans limite sur les processus longs comme les serveurs web.

In general, the LRU cache should only be used when you want to reuse previously computed values. Accordingly, it doesn't make sense to cache functions with side-effects, functions that need to create distinct mutable objects on each call, or impure functions such as `time()` or `random()`.

Exemple d'un cache LRU pour du contenu web statique :

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

Exemple de calcul efficace de la suite de Fibonacci en utilisant un cache pour implémenter la technique de programmation dynamique :

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : L'option *typed* a été ajoutée.

@functools.total_ordering

A partir d'une classe définissant une ou plusieurs méthodes de comparaison riches, ce décorateur de classe fournit le reste. Ceci simplifie l'effort à fournir dans la spécification de toutes les opérations de comparaison riche :

La classe doit définir au moins une de ces méthodes `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`. De plus, la classe doit fournir une méthode `__eq__()`.

Par exemple :

```
@total_ordering
class Student:
    def __is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self.__is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
```

(suite sur la page suivante)

(suite de la page précédente)

```

if not self._is_valid_operand(other):
    return NotImplemented
return ((self.lastname.lower(), self.firstname.lower()) <
        (other.lastname.lower(), other.firstname.lower()))

```

Note : Même si ce décorateur permet de créer des types ordonnables facilement, cela vient avec un *coût* d'exécution et des traces d'exécution complexes pour les méthodes de comparaison dérivées. Si des tests de performances le révèlent comme un goulot d'étranglement, l'implémentation manuelle des six méthodes de comparaison riches résoudra normalement vos problèmes de rapidité.

Nouveau dans la version 3.2.

Modifié dans la version 3.4 : Retourner `NotImplemented` dans les fonction de comparaison sous-jacentes pour les types non reconnus est maintenant supporté.

`functools.partial(func, *args, **keywords)`

Return a new *partial object* which when called will behave like *func* called with the positional arguments *args* and keyword arguments *keywords*. If more arguments are supplied to the call, they are appended to *args*. If additional keyword arguments are supplied, they extend and override *keywords*. Roughly equivalent to :

```

def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc

```

`partial()` est utilisé pour une application de fonction partielle qui « gèle » une portion des arguments et/ou mots-clés d'une fonction donnant un nouvel objet avec une signature simplifiée. Par exemple, `partial()` peut être utilisé pour créer un callable qui se comporte comme la fonction `int()` ou l'argument *base* est deux par défaut :

```

>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18

```

class `functools.partialmethod(func, *args, **keywords)`

Retourne un nouveau descripteur *partialmethod* qui se comporte comme *partial* sauf qu'il est fait pour être utilisé comme une définition de méthode plutôt que d'être appelé directement.

func doit être un *descriptor* ou un callable (les objets qui sont les deux, comme les fonction normales, sont gérés comme des descripteurs).

When *func* is a descriptor (such as a normal Python function, `classmethod()`, `staticmethod()`, `abstractmethod()` or another instance of *partialmethod*), calls to `__get__` are delegated to the underlying descriptor, and an appropriate *partial object* returned as the result.

Quand *func* est un callable non-descripteur, une méthode liée appropriée est créée dynamiquement. Elle se comporte comme une fonction Python normale quand elle est utilisée comme méthode : l'argument *self* sera inséré comme premier argument positionnel, avant les *args* et *keywords* fournis au constructeur *partialmethod*.

Exemple :


```

>>> class Cell(object):
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True

```

Nouveau dans la version 3.4.

`functools.reduce(function, iterable[, initializer])`

Applique *function* avec deux arguments cumulativement aux éléments de *sequence*, de gauche à droite, pour réduire la séquence à une valeur unique. Par exemple, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calcule `((((1+2)+3)+4)+5)`. Le premier argument, *x*, et la valeur de cumul et le deuxième, *y*, est la valeur de mise à jour depuis *sequence*. Si l'argument optionnel *initializer* est présent, il est placé avant les éléments de la séquence dans le calcul, et sert de valeur par défaut quand la séquence est vide. Si *initializer* n'est pas renseigné et que *sequence* ne contient qu'un élément, le premier élément est retourné.

Sensiblement équivalent à :

```

def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value

```

`@functools singledispatch`

Transforme une fonction en une *fonction générique single-dispatch*.

Pour définir une fonction générique, il faut la décorer avec le décorateur `@singledispatch`. Noter que la distribution est effectuée sur le type du premier argument, donc la fonction doit être créée en conséquence :

```

>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)

```

Pour ajouter des surcharges d'implémentation à la fonction, utiliser l'attribut `register()` de la fonction générique. C'est un décorateur, prenant un type en paramètre et décorant une fonction implémentant l'opération pour ce type :

```

>>> @fun.register(int)
... def _(arg, verbose=False):

```

(suite sur la page suivante)

(suite de la page précédente)

```

...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register(list)
... def _(arg, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)

```

Pour permettre l'enregistrement de *lambdas* et de fonctions pré-existantes, l'attribut `register()` peut être utilisé sous forme fonctionnelle :

```

>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)

```

L'attribut `register()` renvoie la fonction non décorée ce qui permet d'empiler les décorateurs, la sérialisation, et la création de tests unitaires pour chaque variante indépendamment :

```

>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False

```

Quand elle est appelée, la fonction générique distribue sur le type du premier argument :

```

>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615

```

Quand il n'y a pas d'implémentation enregistrée pour un type spécifique, son ordre de résolution de méthode est utilisé pour trouver une implémentation plus générique. La fonction originale est décorée avec `@singledispatch` est enregistrée pour le type d'`object`, et elle sera utilisée si aucune implémentation n'est trouvée.

Pour vérifier quelle implémentation la fonction générique choisira pour un type donné, utiliser l'attribut `dispatch()` :

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)      # note: default implementation
<function fun at 0x103fe0000>
```

Pour accéder à toutes les implémentations enregistrées, utiliser l'attribut en lecture seule `registry` :

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
          <class 'decimal.Decimal'>, <class 'list'>,
          <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

Nouveau dans la version 3.4.

`functools.update_wrapper(wrapper, wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)`

Met à jour la fonction `wrapper` pour ressembler à la fonction `wrapped`. Les arguments optionnels sont des tuples pour spécifier quels attributs de la fonction originale sont assignés directement aux attributs correspondants sur la fonction englobante et quels attributs de la fonction englobante sont mis à jour avec les attributs de la fonction originale. Les valeurs par défaut de ces arguments sont les constantes au niveau du module `WRAPPER_ASSIGNMENTS` (qui assigne `__module__`, `__name__`, `__qualname__`, `__annotations__` et `__doc__`, la chaîne de documentation, depuis la fonction englobante) et `WRAPPER_UPDATES` (qui met à jour le `__dict__` de la fonction englobante, c'est-à-dire le dictionnaire de l'instance).

Pour autoriser l'accès à la fonction originale pour l'introspection ou à d'autres fins (par ex. outrepasser l'accès à un décorateur de cache comme `lru_cache()`), cette fonction ajoute automatiquement un attribut `__wrapped__` qui référence la fonction englobée.

La principale utilisation de cette fonction est dans les *décorateurs* qui renvoient une nouvelle fonction. Si la fonction créée n'est pas mise à jour, ses métadonnées refléteront sa définition dans le décorateur, au lieu de la définition originale, métadonnées souvent bien moins utiles.

`update_wrapper()` peut être utilisé avec des appelables autres que des fonctions. Tout attribut défini dans `assigned` ou `updated` qui ne sont pas l'objet englobé sont ignorés (cette fonction n'essaiera pas de les définir dans la fonction englobante). `AttributeError` est toujours levée si la fonction englobante elle-même a des attributs non existants dans `updated`.

Nouveau dans la version 3.2 : Ajout automatique de l'attribut `__wrapped__`.

Nouveau dans la version 3.2 : Copie de l'attribut `__annotations__` par défaut.

Modifié dans la version 3.2 : Les attributs manquants ne lèvent plus d'exception `AttributeError`.

Modifié dans la version 3.4 : L'attribut `__wrapped__` renvoie toujours la fonction englobée, même si cette fonction définit un attribut `__wrapped__`. (voir [bpo-17482](#))

`@functools.wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)`

Ceci est une fonction d'aide pour appeler `update_wrapper()` comme décorateur de fonction lors de la définition d'une fonction englobante. C'est équivalent à `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)`. Par exemple :

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
... 
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

Sans l'utilisation de cette usine à décorateur, le nom de la fonction d'exemple aurait été 'wrapper', et la chaîne de documentation de la fonction `example()` originale aurait été perdue.

10.2.1 Objets *partial*

Les objets *partial* sont des objets appelables créés par *partial()*. Ils ont trois attributs en lecture seule :

partial.**func**

Un objet ou une fonction callable. Les appels à l'objet *partial* seront transmis à *func* avec les nouveaux arguments et mots-clés.

partial.**args**

Les arguments positionnels qui seront ajoutés avant les arguments fournis lors de l'appel d'un objet *partial*.

partial.**keywords**

Les arguments nommés qui seront fournis quand l'objet *partial* est appelé.

Les objets *partial* sont comme des objets *function* de par le fait qu'il sont appelables, référençables, et peuvent avoir des attributs. Il y a cependant des différences importantes. Par exemple, les attributs `__name__` et `__doc__` ne sont pas créés automatiquement. De plus, les objets *partial* définis dans les classes se comportent comme des méthodes statiques et ne se transforment pas en méthodes liées durant la recherche d'attributs dans l'instance.

10.3 *operator* — Opérateurs standards en tant que fonctions

Code source : [Lib/operator.py](#)

Le module *operator* fournit un ensemble de fonctions correspondant aux opérateurs natifs de Python. Par exemple, `operator.add(x, y)` correspond à l'expression `x+y`. Les noms de la plupart de ces fonctions sont ceux utilisés par les méthodes spéciales, sans les doubles tirets bas. Pour assurer la rétrocompatibilité, la plupart de ces noms ont une variante *avec* les doubles tirets bas ; la forme simple est cependant à privilégier pour des raisons de clarté.

Les fonctions sont divisées en différentes catégories selon l'opération effectuée : comparaison entre objets, opérations logiques, opérations mathématiques ou opérations sur séquences.

Les fonctions de comparaison s'appliquent à tous les objets, et leur nom vient des opérateurs de comparaison qu'elles implémentent :

```
operator.lt(a, b)
operator.le(a, b)
operator.eq(a, b)
operator.ne(a, b)
```

```

operator.ge(a, b)
operator.gt(a, b)
operator.lt(a, b)
operator.le(a, b)
operator.eq(a, b)
operator.ne(a, b)
operator.ge(a, b)
operator.gt(a, b)

```

Effectue une « comparaison riche » entre *a* et *b*. Plus précisément, `lt(a, b)` équivaut à `a < b`, `le(a, b)` équivaut à `a <= b`, `eq(a, b)` équivaut à `a == b`, `ne(a, b)` équivaut à `a != b`, `gt(a, b)` équivaut à `a > b` et `ge(a, b)` équivaut à `a >= b`. Notez que ces fonctions peuvent renvoyer n'importe quelle valeur, convertible ou non en booléen. Voir `comparisons` pour plus d'informations sur les méthodes de comparaison riches.

En général, les opérations logiques s'appliquent aussi à tous les objets et implémentent les tests de vérité, d'identité et les opérations booléennes :

```

operator.not_(obj)
operator.__not__(obj)

```

Renvoie le résultat de `not obj`. (Notez qu'il n'existe pas de méthode `__not__()` pour les instances d'objet ; seul le cœur de l'interpréteur définit cette opération. Le résultat dépend des méthodes `__bool__()` et `__len__()`.)

```
operator.truth(obj)
```

Renvoie `True` si *obj* est vrai, et `False` dans le cas contraire. Équivaut à utiliser le constructeur de `bool`.

```
operator.is_(a, b)
```

Renvoie `a is b`. Vérifie si les deux paramètres sont le même objet.

```
operator.is_not(a, b)
```

Renvoie `a is not b`. Vérifie si les deux paramètres sont deux objets distincts.

Les opérations mathématiques ou bit à bit sont les plus nombreuses :

```

operator.abs(obj)
operator.__abs__(obj)

```

Renvoie la valeur absolue de *obj*.

```

operator.add(a, b)
operator.__add__(a, b)

```

Renvoie `a + b` où *a* et *b* sont des nombres.

```

operator.and_(a, b)
operator.__and__(a, b)

```

Renvoie le *et* bit à bit de *a* et *b*.

```

operator.floordiv(a, b)
operator.__floordiv__(a, b)

```

Renvoie `a // b`.

```

operator.index(a)
operator.__index__(a)

```

Renvoie *a* converti en entier. Équivaut à `a.__index__()`.

```

operator.inv(obj)
operator.invert(obj)
operator.__inv__(obj)
operator.__invert__(obj)

```

Renvoie l'inverse bit à bit du nombre *obj*. Équivaut à `~obj`.

```
operator.lshift(a, b)
```

`operator.lshift(a, b)`

Renvoie le décalage de *b* bits vers la gauche de *a*.

`operator.mod(a, b)`

`operator.__mod__(a, b)`

Renvoie “*a* % *b*”.

`operator.mul(a, b)`

`operator.__mul__(a, b)`

Renvoie *a* * *b* où *a* et *b* sont des nombres.

`operator.matmul(a, b)`

`operator.__matmul__(a, b)`

Renvoie *a* @ *b*.

Nouveau dans la version 3.5.

`operator.neg(obj)`

`operator.__neg__(obj)`

Renvoie l'opposé de *obj* (-*obj*).

`operator.or_(a, b)`

`operator.__or__(a, b)`

Renvoie le *ou* bit à bit de *a* et *b*.

`operator.pos(obj)`

`operator.__pos__(obj)`

Renvoie la valeur positive de *obj* (+*obj*).

`operator.pow(a, b)`

`operator.__pow__(a, b)`

Renvoie *a* ** *b* où *a* et *b* sont des nombres.

`operator.rshift(a, b)`

`operator.__rshift__(a, b)`

Renvoie le décalage de *b* bits vers la droite de *a*.

`operator.sub(a, b)`

`operator.__sub__(a, b)`

Renvoie *a* - *b*.

`operator.truediv(a, b)`

`operator.__truediv__(a, b)`

Renvoie *a* / *b* où 2/3 est 0.66 et non 0. Appelée aussi division « réelle ».

`operator.xor(a, b)`

`operator.__xor__(a, b)`

Renvoie le *ou* exclusif bit à bit de *a* et *b*.

Les opérations sur séquences (et pour certaines, sur correspondances) sont :

`operator.concat(a, b)`

`operator.__concat__(a, b)`

Renvoie *a* + *b* où *a* et *b* sont des séquences.

`operator.contains(a, b)`

`operator.__contains__(a, b)`

Renvoie le résultat du test *b* in *a*. Notez que les opérandes sont inversées.

`operator.countOf(a, b)`

Renvoie le nombre d'occurrences de *b* dans *a*.

`operator.delitem(a, b)`

`operator.__delitem__(a, b)`
 Renvoie la valeur de *a* à l'indice *b*.

`operator.getitem(a, b)`
`operator.__getitem__(a, b)`
 Renvoie la valeur de *a* à l'indice *b*.

`operator.indexof(a, b)`
 Renvoie l'indice de la première occurrence de *b* dans *a*.

`operator.setitem(a, b, c)`
`operator.__setitem__(a, b, c)`
 Affecte *c* dans *a* à l'indice *b*.

`operator.length_hint(obj, default=0)`
 Renvoie une estimation de la taille de l'objet *o*. Tente d'abord de renvoyer la taille réelle, puis une estimation en appelant `object.__length_hint__()`, ou sinon la valeur par défaut.
 Nouveau dans la version 3.4.

Le module `operator` définit aussi des fonctions pour la recherche générique d'attributs ou d'objets. Elles sont particulièrement utiles pour construire rapidement des accesseurs d'attributs à passer en paramètre à `map()`, `sorted()`, `itertools.groupby()` ou à toute autre fonction prenant une fonction en paramètre.

`operator.attrgetter(attr)`
`operator.attrgetter(*attrs)`

Renvoie un objet callable qui récupère *attr* de son opérande. Si plus d'un attribut est demandé, renvoie un n-uplet d'attributs. Les noms des attributs peuvent aussi comporter des points. Par exemple :

- Avec `f = attrgetter('name')`, l'appel `f(b)` renvoie `b.name`.
- Avec `f = attrgetter('name', 'date')`, l'appel `f(b)` renvoie `(b.name, b.date)`.
- Après `f = attrgetter('name.first', 'name.last')`, l'appel `f(b)` renvoie `(b.name.first, b.name.last)`.

Équivalent à :

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

`operator.itemgetter(item)`
`operator.itemgetter(*items)`

Renvoie un objet callable qui récupère *item* de l'opérande en utilisant la méthode `__getitem__()`. Si plusieurs *item* sont passés en paramètre, renvoie un n-uplet des valeurs récupérées. Par exemple :

- Avec `f = itemgetter(2)`, `f(r)` renvoie `r[2]`.
- Avec `g = itemgetter(2, 5, 3)`, `g(r)` renvoie `(r[2], r[5], r[3])`.

Équivalent à :

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

Les *items* en entrée peuvent être de n'importe quel type tant que celui-ci est géré par la méthode `__getitem__()` de l'opérande. Les dictionnaires acceptent toute valeur hachable. Les listes, n-uplets et chaînes de caractères acceptent un index ou une tranche :

```
>>> itemgetter(1)('ABCDEFGH')
'B'
>>> itemgetter(1,3,5)('ABCDEFGH')
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFGH')
'CDEFGH'
```

Exemple d'utilisation de `itemgetter()` pour récupérer des champs spécifiques d'un n-uplet :

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

`operator.methodcaller(name[, args...])`

Renvoie un objet callable qui appelle la méthode *name* de son opérande. Si des paramètres supplémentaires et/ou des paramètres nommés sont donnés, ils seront aussi passés à la méthode. Par exemple :

- Avec `f = methodcaller('name')`, `f(b)` renvoie `b.name()`.
- Avec `f = methodcaller('name', 'foo', bar=1)`, `f(b)` renvoie `b.name('foo', bar=1)`.

Équivalent à :

```
def methodcaller(name, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 Correspondances entre opérateurs et fonctions

Le tableau montre la correspondance entre les symboles des opérateurs Python et les fonctions du module `operator`.

Opération	Syntaxe	Fonction
Addition	<code>a + b</code>	<code>add(a, b)</code>
Concaténation	<code>seq1 + seq2</code>	<code>concat(seq1, seq2)</code>
Test d'inclusion	<code>obj in seq</code>	<code>contains(seq, obj)</code>
Division	<code>a / b</code>	<code>truediv(a, b)</code>
Division	<code>a // b</code>	<code>floordiv(a, b)</code>
Et bit à bit	<code>a & b</code>	<code>and_(a, b)</code>

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Opération	Syntaxe	Fonction
<i>Ou exclusif</i> bit à bit	<code>a ^ b</code>	<code>xor(a, b)</code>
Inversion bit à bit	<code>~ a</code>	<code>invert(a)</code>
<i>Ou</i> bit à bit	<code>a b</code>	<code>or_(a, b)</code>
Exponentiation	<code>a ** b</code>	<code>pow(a, b)</code>
Identité	<code>a is b</code>	<code>is_(a, b)</code>
Identité	<code>a is not b</code>	<code>is_not(a, b)</code>
Affectation par index	<code>obj[k] = v</code>	<code>setitem(obj, k, v)</code>
Suppression par index	<code>del obj[k]</code>	<code>delitem(obj, k)</code>
Indexation	<code>obj[k]</code>	<code>getitem(obj, k)</code>
Décalage bit à bit gauche	<code>a << b</code>	<code>lshift(a, b)</code>
Modulo	<code>a % b</code>	<code>mod(a, b)</code>
Multiplication	<code>a * b</code>	<code>mul(a, b)</code>
Multiplication matricielle	<code>a @ b</code>	<code>matmul(a, b)</code>
Opposé	<code>- a</code>	<code>neg(a)</code>
Négation (logique)	<code>not a</code>	<code>not_(a)</code>
Valeur positive	<code>+ a</code>	<code>pos(a)</code>
Décalage bit à bit droite	<code>a >> b</code>	<code>rshift(a, b)</code>
Affectation par tranche	<code>seq[i:j] = values</code>	<code>setitem(seq, slice(i, j), values)</code>
Suppression par tranche	<code>del seq[i:j]</code>	<code>delitem(seq, slice(i, j))</code>
Tranche	<code>seq[i:j]</code>	<code>getitem(seq, slice(i, j))</code>
Formatage de chaînes de caractères	<code>s % obj</code>	<code>mod(s, obj)</code>
Soustraction	<code>a - b</code>	<code>sub(a, b)</code>
Test de véracité	<code>obj</code>	<code>truth(obj)</code>
Ordre	<code>a < b</code>	<code>lt(a, b)</code>
Ordre	<code>a <= b</code>	<code>le(a, b)</code>
Égalité	<code>a == b</code>	<code>eq(a, b)</code>
Inégalité	<code>a != b</code>	<code>ne(a, b)</code>
Ordre	<code>a >= b</code>	<code>ge(a, b)</code>
Ordre	<code>a > b</code>	<code>gt(a, b)</code>

10.3.2 Inplace Operators

Beaucoup d'opérations ont une version travaillant « en-place ». Les fonctions listées ci-dessous fournissent un accès plus direct aux opérateurs en-place que la syntaxe Python habituelle ; par exemple, l'expression `statement x += y` équivaut à `x = operator.iadd(x, y)`. Autrement dit, l'expression `z = operator.iadd(x, y)` équivaut à l'expression composée `z = x; z += y`.

Dans ces exemples, notez que lorsqu'une méthode en-place est appelée, le calcul et l'affectation sont effectués en deux étapes distinctes. Les fonctions en-place de la liste ci-dessous ne font que la première, en appelant la méthode en-place. La seconde étape, l'affectation, n'est pas effectuée.

Pour des paramètres non-mutables comme les chaînes de caractères, les nombres et les n-uplets, la nouvelle valeur est calculée, mais pas affectée à la variable d'entrée :

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

For mutable targets such as lists and dictionaries, the inplace method will perform the update, so no subsequent assignment

is necessary :

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

`operator.iadd(a, b)`

`operator.__iadd__(a, b)`

`a = iadd(a, b)` équivaut à `a += b`.

`operator.iand(a, b)`

`operator.__iand__(a, b)`

`a = iand(a, b)` équivaut à `a &= b`.

`operator.iconcat(a, b)`

`operator.__iconcat__(a, b)`

`a = iconcat(a, b)` équivaut à `a += b` où *a* et *b* sont des séquences.

`operator.ifloordiv(a, b)`

`operator.__ifloordiv__(a, b)`

`a = ifloordiv(a, b)` équivaut à `a //= b`.

`operator.ilshift(a, b)`

`operator.__ilshift__(a, b)`

`a = ilshift(a, b)` équivaut à `a <= b`.

`operator.imod(a, b)`

`operator.__imod__(a, b)`

`a = imod(a, b)` équivaut à `a %= b`.

`operator.imul(a, b)`

`operator.__imul__(a, b)`

`a = imul(a, b)` équivaut à `a *= b`.

`operator.imatmul(a, b)`

`operator.__imatmul__(a, b)`

`a = imatmul(a, b)` équivaut à `a @= b`.

Nouveau dans la version 3.5.

`operator.ior(a, b)`

`operator.__ior__(a, b)`

`a = ior(a, b)` équivaut à `a |= b`.

`operator.ipow(a, b)`

`operator.__ipow__(a, b)`

`a = ipow(a, b)` équivaut à `a **= b`.

`operator.irshift(a, b)`

`operator.__irshift__(a, b)`

`a = irshift(a, b)` équivaut à `a >= b`.

`operator.isub(a, b)`

`operator.__isub__(a, b)`

`a = isub(a, b)` équivaut à `a -= b`.

`operator.itruediv(a, b)`

`operator.__itruediv__(a, b)`

`a = itrueidiv(a, b)` équivaut à `a /= b`.

```
operator. ixor(a, b)  
operator. __ixor__(a, b)  
    a = ixor(a, b) équivaut à  $a \wedge= b$ .
```

Accès aux Fichiers et aux Dossiers

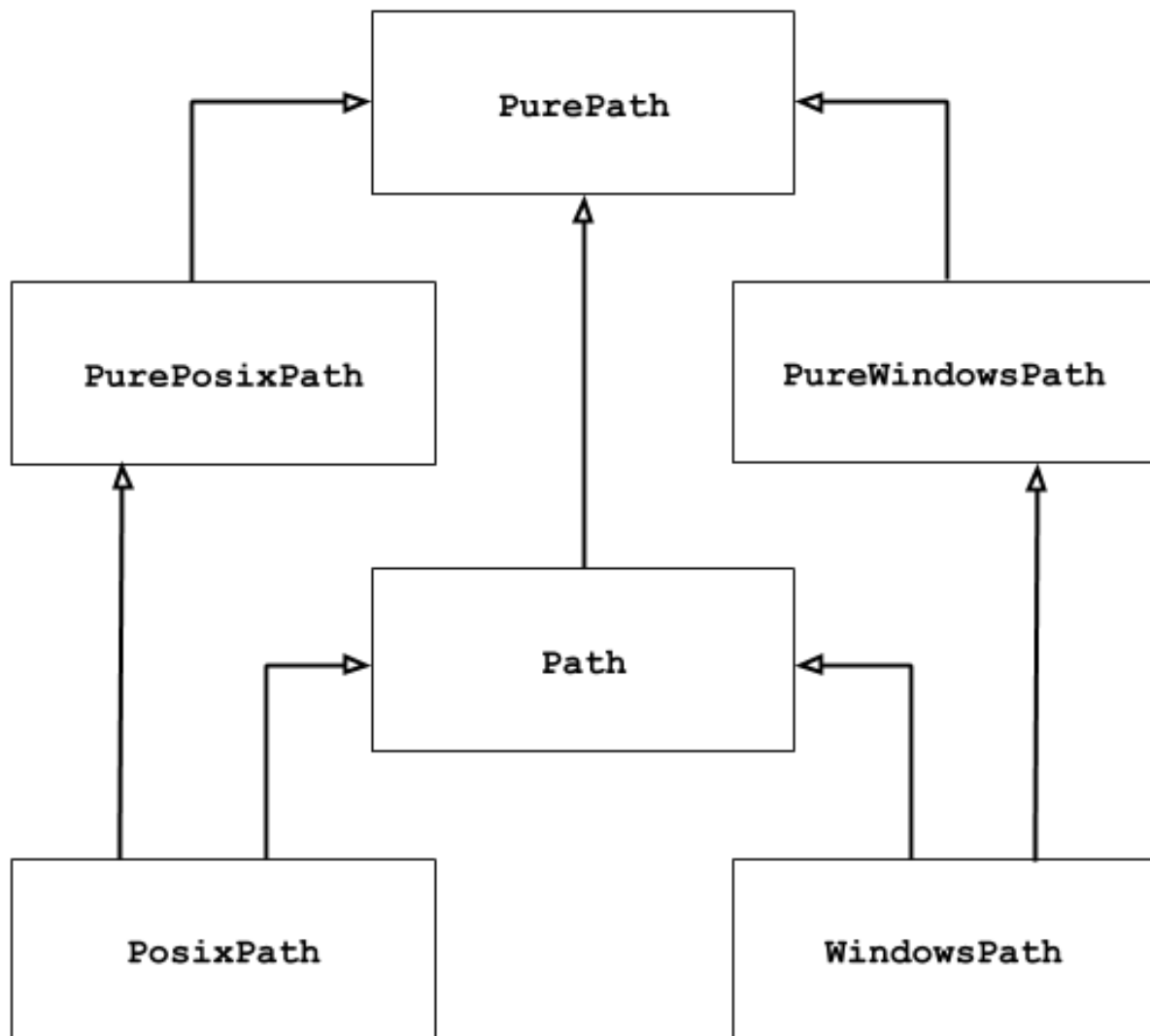
Les modules décrits dans ce chapitre servent à accéder aux fichiers et aux dossiers. Des modules, par exemple, pour lire les propriétés des fichiers, manipuler des chemins de manière portable, et créer des fichiers temporaires. La liste exhaustive des modules de ce chapitre est :

11.1 `pathlib` — Chemins de système de fichiers orientés objet

Nouveau dans la version 3.4.

Code source : [Lib/pathlib.py](#)

Ce module offre des classes représentant le système de fichiers avec la sémantique appropriée pour différents systèmes d'exploitation. Les classes de chemins sont divisées en *chemins purs*, qui fournissent purement du calcul sans entrées/sorties, et *chemins concrets*, qui héritent des chemins purs et fournissent également les opérations d'entrées/sorties.



Si vous n’avez jamais utilisé ce module précédemment, ou si vous n’êtes pas sûr de quelle classe est faite pour votre tâche, `Path` est très certainement ce dont vous avez besoin. Elle instancie un *chemin concret* pour la plateforme sur laquelle s’exécute le code.

Les chemins purs sont utiles dans certains cas particuliers ; par exemple :

1. Si vous voulez manipuler des chemins Windows sur une machine Unix (ou vice versa). Vous ne pouvez pas instancier un `WindowsPath` quand vous êtes sous Unix, mais vous pouvez instancier `PureWindowsPath`.
2. Vous voulez être sûr que votre code manipule des chemins sans réellement accéder au système d’exploitation. Dans ce cas, instancier une de ces classes pures peut être utile puisqu’elle ne possède tout simplement aucune opération permettant d’accéder au système d’exploitation.

Voir aussi :

PEP 428 : Le module `pathlib` – chemins de système de fichiers orientés objet.

Voir aussi :

Pour de la manipulation de chemins bas-niveau avec des chaînes de caractères, vous pouvez aussi utiliser le module `os.path`.

11.1.1 Utilisation basique

Importer la classe principale :

```
>>> from pathlib import Path
```

Lister les sous-dossiers :

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

Lister les fichiers source Python dans cette arborescence de dossiers :

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

Naviguer à l'intérieur d'une arborescence de dossiers :

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

Récupérer les propriétés de chemin :

```
>>> q.exists()
True
>>> q.is_dir()
False
```

Ouvrir un fichier :

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 Chemins purs

Les objets chemins purs fournissent les opérations de gestion de chemin qui n'accèdent pas réellement au système de fichiers. Il y a trois façons d'accéder à ces classes que nous appelons aussi *familles* :

class `pathlib.PurePath` (**pathsegments*)

Une classe générique qui représente la famille de chemin du système (l'instancier crée soit un *PurePosixPath* soit un *PureWindowsPath*) :

```
>>> PurePath('setup.py')           # Running on a Unix machine
PurePosixPath('setup.py')
```

Chaque élément de *pathsegments* peut soit être une chaîne de caractères représentant un segment de chemin, un objet implémentant l'interface *os.PathLike* qui renvoie une chaîne de caractères, soit un autre objet chemin :

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

Quand *pathsegments* est vide, le dossier courant est utilisé :

```
>>> PurePath()
PurePosixPath('.')
```

Quand plusieurs chemins absolus sont fournis, le dernier est pris comme ancre (recopiant le comportement de *os.path.join()*) :

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

Cependant, dans un chemin Windows, changer la racine locale ne supprime pas la précédente configuration de lecteur :

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Les points et slashes malencontreux sont supprimés, mais les doubles points ('..') ne le sont pas, puisque cela changerait la signification du chemin dans le cas de liens symboliques :

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(une analyse naïve considérerait *PurePosixPath('foo/../bar')* équivalent à *PurePosixPath('bar')*, ce qui est faux si *foo* est un lien symbolique vers un autre dossier)

Les objets chemins purs implémentent l'interface *os.PathLike*, leur permettant d'être utilisés n'importe où l'interface est acceptée.

Modifié dans la version 3.6 : Ajout de la gestion de l'interface *os.PathLike*.

class *pathlib.PurePosixPath* (**pathsegments*)

Une sous-classe de *PurePath*, cette famille de chemin représente les chemins de systèmes de fichiers en dehors des chemins Windows :

```
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

pathsegments est spécifié de manière similaire à *PurePath*.

class *pathlib.PureWindowsPath* (**pathsegments*)

Une sous-classe de *PurePath*, cette famille de chemin représente les chemins de systèmes de fichiers Windows :

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
```

pathsegments est spécifié de manière similaire à *PurePath*.

Sans tenir compte du système sur lequel vous êtes, vous pouvez instancier toutes ces classes, puisqu'elles ne fournissent aucune opération qui appelle le système d'exploitation.

Propriétés générales

Les chemins sont immuables et hachables. Les chemins d'une même famille sont comparables et ordonnables. Ces propriétés respectent l'ordre lexicographique défini par la famille :

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

Les chemins de différentes familles ne sont pas égaux et ne peuvent être ordonnés :

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPath'
↪ '
```

Opérateurs

L'opérateur slash aide à créer les chemins enfants, de manière similaire à `os.path.join()` :

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
```

Un objet chemin peut être utilisé n'importe où un objet implémentant `os.PathLike` est accepté :

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

La représentation d'un chemin en chaîne de caractères est celle du chemin brut du système de fichiers lui-même (dans sa forme native, i.e. avec des antislashes sous Windows), et que vous pouvez passer à n'importe quelle fonction prenant un chemin en tant que chaîne de caractères :

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

De manière similaire, appeler `bytes` sur un chemin donne le chemin brut du système de fichiers en tant que bytes, tel qu'encodé par `os.fsencode()` :

```
>>> bytes(p)
b'/etc'
```

Note : Appeler *bytes* est seulement recommandé sous Unix. Sous Windows, la forme Unicode est la représentation canonique des chemins du système de fichiers.

Accéder aux parties individuelles

Pour accéder aux parties individuelles (composantes) d'un chemin, utilisez les propriétés suivantes :

`PurePath.parts`

Un tuple donnant accès aux différentes composantes du chemin :

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

(notez comme le lecteur et la racine locale sont regroupés en une seule partie)

Méthodes et propriétés

Les chemins purs fournissent les méthodes et propriétés suivantes :

`PurePath.drive`

Une chaîne représentant la lettre du lecteur ou le nom, s'il y en a un :

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

Les partages UNC sont aussi considérés comme des lecteurs :

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\\\host\\share'
```

`PurePath.root`

Une chaîne de caractères représentant la racine (locale ou globale), s'il y en a une :

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

Les partages UNC ont toujours une racine :

```
>>> PureWindowsPath('//host/share').root
'\\'
```

PurePath.anchor

La concaténation du lecteur et de la racine :

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
 '/'
>>> PureWindowsPath('//host/share').anchor
 '\\\\host\\share\\'
```

PurePath.parents

Une séquence immuable fournissant accès aux ancêtres logiques du chemin :

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

PurePath.parent

Le parent logique du chemin :

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

Vous ne pouvez pas aller au-delà d'une ancre, ou d'un chemin vide :

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
>>> p = PurePosixPath('')
>>> p.parent
PurePosixPath('')
```

Note : C'est une opération purement lexicale, d'où le comportement suivant :

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

Si vous voulez parcourir un chemin arbitraire du système de fichiers, il est recommandé de d'abord appeler `Path.resolve()` de manière à résoudre les liens symboliques et éliminer les composantes « .. ».

PurePath.name

Une chaîne représentant la composante finale du chemin, en excluant le lecteur et la racine, si présent :

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

Les noms de lecteur UNC ne sont pas pris en compte :

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.suffix

L'extension du fichier de la composante finale, si présente :

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

PurePath.suffixes

Une liste des extensions du chemin de fichier :

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

PurePath.stem

La composante finale du chemin, sans son suffixe :

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.as_posix()

Renvoie une représentation en chaîne de caractères du chemin avec des slashes (/) :

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.as_uri()

Représente le chemin en tant qu'URI de fichier. *ValueError* est levée si le chemin n'est pas absolu.

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = PureWindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

PurePath.is_absolute()

Renvoie si le chemin est absolu ou non. Un chemin est considéré absolu s'il a une racine et un lecteur (si la famille

le permet) :

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

`PurePath.is_reserved()`

Avec `PureWindowsPath`, renvoie `True` si le chemin est considéré réservé sous Windows, `False` sinon. Avec `PurePosixPath`, `False` est systématiquement renvoyé.

```
>>> PureWindowsPath('nul').is_reserved()
True
>>> PurePosixPath('nul').is_reserved()
False
```

Les appels au système de fichier sur des chemins réservés peuvent échouer mystérieusement ou avoir des effets inattendus.

`PurePath.joinpath(*other)`

Appeler cette méthode équivaut à combiner le chemin avec chacun des arguments *other* :

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

`PurePath.match(pattern)`

Fait correspondre ce chemin avec le motif (*glob pattern*) fourni. Renvoie `True` si la correspondance a réussi, `False` sinon.

Si *pattern* est relatif, le chemin peut être soit relatif, soit absolu, et la correspondance est faite à partir de la droite :

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

Si *pattern* est absolu, le chemin doit être absolu, et la correspondance doit être totale avec le chemin :

```
>>> PurePath('/a.py').match('/*.py')
True
>>> PurePath('a/b.py').match('/*.py')
False
```

Comme avec les autres méthodes, la casse est prise en compte :

```
>>> PureWindowsPath('b.py').match('*.PY')
True
```

`PurePath.relative_to(*other)`

Calcule une version du chemin en relatif au chemin représenté par *other*. Si c'est impossible, `ValueError` est levée :

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 694, in relative_to
    .format(str(self), str(formatted)))
ValueError: '/etc/passwd' does not start with '/usr'
```

`PurePath.with_name(name)`

Renvoie un nouveau chemin avec *name* changé. Si le chemin original n'a pas de nom, `ValueError` est levée :

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

`PurePath.with_suffix(suffix)`

Return a new path with the *suffix* changed. If the original path doesn't have a suffix, the new *suffix* is appended instead. If the *suffix* is an empty string, the original suffix is removed :

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

11.1.3 Chemins concrets

Les chemins concrets sont des sous-classes des chemins purs. En plus des opérations fournies par ces derniers, ils fournissent aussi des méthodes pour faire appel au système sur des objets chemin. Il y a trois façons d'instancier des chemins concrets :

class `pathlib.Path` (**pathsegments*)

Une sous-classe de *PurePath*, cette classe représente les chemins concrets d'une famille de chemins de système de fichiers (l'instancier créé soit un *PosixPath*, soit un *WindowsPath*) :

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments est spécifié de manière similaire à *PurePath*.

class `pathlib.PosixPath` (**pathsegments*)

Une sous-classe de *Path* et *PurePosixPath*, cette classe représente les chemins concrets de systèmes de fichiers non Windows :

```
>>> PosixPath('/etc')
PosixPath('/etc')
```

pathsegments est spécifié de manière similaire à *PurePath*.

class `pathlib.WindowsPath` (**pathsegments*)

Une sous-classe de *Path* et *PureWindowsPath*, cette classe représente les chemins concrets de systèmes de fichiers Windows :

```
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```

pathsegments est spécifié de manière similaire à *PurePath*.

Vous ne pouvez instancier la classe de la famille qui correspond à votre système (permettre des appels au système pour des familles de chemins non compatible pourrait mener à des bogues ou à des pannes de votre application) :

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

Méthodes

Les chemins concrets fournissent les méthodes suivantes en plus des méthodes des chemins purs. Beaucoup de ces méthodes peuvent lever `OSError` si un appel au système échoue (par exemple car le chemin n'existe pas) :

classmethod `Path.cwd()`

Renvoie un nouveau chemin représentant le dossier courant (comme renvoyé par `os.getcwd()`) :

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

classmethod `Path.home()`

Renvoie un nouveau chemin représentant le dossier d'accueil de l'utilisateur (comme renvoyé par `os.path.expanduser()` avec la construction `~`) :

```
>>> Path.home()
PosixPath('/home/antoine')
```

Nouveau dans la version 3.5.

`Path.stat()`

Renvoie les informations à propos de ce chemin (de manière similaire à `os.stat()`). Le résultat est récupéré à chaque appel à cette méthode.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

`Path.chmod(mode)`

Change le mode et les permissions du fichier, comme `os.chmod()` :

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

`Path.exists()`

Si le chemin pointe sur un fichier ou dossier existant :

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

Note : Si le chemin pointe sur un lien symbolique, `exists()` renvoie si le lien symbolique *pointe vers* un fichier ou un dossier existant.

`Path.expanduser()`

Renvoie un nouveau chemin avec les résolutions des constructions `~` et `~user`, comme retourné par `os.path.expanduser()` :

```
>>> p = PosixPath('~ /films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

Nouveau dans la version 3.5.

Path.glob(pattern)

Globalise le *pattern* fourni dans le dossier représenté par ce chemin, donnant tous les fichiers correspondants (de n'importe quelle sorte) :

```
>>> sorted(Path('.').glob('* .py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('* /* .py'))
[PosixPath('docs/conf.py')]
```

Le motif « `**` » signifie que « ce dossier et ses sous-dossiers, récursivement ». En d'autres mots, il active la récursivité de la globalisation :

```
>>> sorted(Path('.').glob('* /* .py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Note : Utiliser le motif « `**` » dans de grandes arborescences de dossier peut consommer une quantité de temps démesurée.

Path.group()

Renvoie le nom du groupe auquel appartient le fichier. `KeyError` est levée si l'identifiant de groupe du fichier n'est pas trouvé dans la base de données système.

Path.is_dir()

Renvoie `True` si le chemin pointe vers un dossier (ou un lien symbolique pointant vers un dossier), `False` s'il pointe vers une autre sorte de fichier.

`False` est aussi renvoyé si le chemin n'existe pas ou est un lien symbolique cassé ; d'autres erreurs (telles que les erreurs de permission) sont propagées.

Path.is_file()

Renvoie `True` si le chemin pointe vers un fichier normal (ou un lien symbolique pointant vers un fichier normal), `False` s'il pointe vers une autre sorte de fichier.

`False` est aussi renvoyé si le chemin n'existe pas ou est un lien symbolique cassé ; d'autres erreurs (telles que les erreurs de permission) sont propagées.

Path.is_symlink()

Renvoie `True` si le chemin pointe sur un lien symbolique, `False` sinon.

`False` est aussi renvoyé si le chemin n'existe pas ; d'autres erreurs (telles que les erreurs de permission) sont propagées.

Path.is_socket()

Renvoie `True` si le chemin pointe vers un connecteur Unix (ou un lien symbolique pointant vers un connecteur Unix), `False` s'il pointe vers une autre sorte de fichier.

`False` est aussi renvoyé si le chemin n'existe pas ou est un lien symbolique cassé ; d'autres erreurs (telles que les erreurs de permission) sont propagées.

`Path.is_fifo()`

Renvoie `True` si le chemin pointe vers une FIFO (ou un lien symbolique pointant vers une FIFO), `False` s'il pointe vers une autre sorte de fichier.

`False` est aussi renvoyé si le chemin n'existe pas ou est un lien symbolique cassé; d'autres erreurs (telles que les erreurs de permission) sont propagées.

`Path.is_block_device()`

Renvoie `True` si le chemin pointe vers un périphérique (ou un lien symbolique pointant vers un périphérique), `False` s'il pointe vers une autre sorte de fichier.

`False` est aussi renvoyé si le chemin n'existe pas ou est un lien symbolique cassé; d'autres erreurs (telles que les erreurs de permission) sont propagées.

`Path.is_char_device()`

Renvoie `True` si le chemin pointe vers un périphérique à caractères (ou un lien symbolique pointant vers un périphérique à caractères), `False` s'il pointe vers une autre sorte de fichier.

`False` est aussi renvoyé si le chemin n'existe pas ou est un lien symbolique cassé; d'autres erreurs (telles que les erreurs de permission) sont propagées.

`Path.iterdir()`

Quand le chemin pointe vers un dossier, donne les chemins du contenu du dossier :

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

`Path.lchmod(mode)`

Comme `Path.chmod()`, mais, si le chemin pointe vers un lien symbolique, le mode du lien symbolique est changé plutôt que celui de sa cible.

`Path.lstat()`

Comme `Path.stat()`, mais, si le chemin pointe vers un lien symbolique, renvoie les informations du lien symbolique plutôt que celui de sa cible.

`Path.mkdir(mode=0o777, parents=False, exist_ok=False)`

Créer un nouveau dossier au chemin fourni. Si `mode` est fourni, il est combiné avec la valeur de l'`umask` du processus pour déterminer le mode de fichier et les droits d'accès. Si le chemin existe déjà, `FileExistsError` est levée.

Si `parents` est vrai, chaque parent de ce chemin est créé si besoin; ils sont créés avec les permissions par défaut sans prendre en compte `mode` (reproduisant la commande POSIX `mkdir -p`).

Si `parents` est faux (valeur par défaut), un parent manquant lève `FileNotFoundError`.

Si `exist_ok` est faux (valeur par défaut), `FileExistsError` est levé si le dossier cible existe déjà.

Si `exist_ok` est vrai, les exceptions `FileExistsError` seront ignorées (même comportement que la commande POSIX `mkdir -p`), mais seulement si le dernier segment de chemin existe et n'est pas un dossier.

Modifié dans la version 3.5 : Le paramètre `exist_ok` a été ajouté.

`Path.open(mode='r', buffering=-1, encoding=None, errors=None, newline=None)`

Ouvre le fichier pointé par le chemin, comme la fonction native `open()` le fait :

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

`Path.owner()`

Renvoie le nom de l'utilisateur auquel appartient le fichier. `KeyError` est levée si l'identifiant utilisateur du fichier n'est pas trouvé dans la base de données du système.

`Path.read_bytes()`

Renvoie le contenu binaire du fichier pointé en tant que bytes :

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Nouveau dans la version 3.5.

`Path.read_text(encoding=None, errors=None)`

Renvoie le contenu décodé du fichier pointé en tant que chaîne de caractères :

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

The file is opened and then closed. The optional parameters have the same meaning as in `open()`.

Nouveau dans la version 3.5.

`Path.rename(target)`

Renomme ce fichier ou dossier vers la cible *target* fournie. Sur Unix, si *target* existe et que c'est un fichier, il sera remplacé silencieusement si l'utilisateur a la permission. *target* peut être soit une chaîne de caractères, soit un autre chemin :

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
>>> target.open().read()
'some text'
```

`Path.replace(target)`

Renomme ce fichier ou dossier vers la cible *target* fournie. Si *target* pointe sur un fichier ou un dossier existant, il sera remplacé de manière inconditionnelle.

`Path.resolve(strict=False)`

Rend le chemin absolu, résolvant les liens symboliques. Un nouveau chemin est renvoyé :

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

Les composantes « `..` » sont aussi éliminées (c'est la seule méthode pour le faire) :

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

Si le chemin n'existe pas et que *strict* est *True*, *FileNotFoundError* est levée. Si *strict* est *False*, le chemin est résolu aussi loin que possible et le reste potentiel est ajouté à la fin sans vérifier s'il existe. Si une boucle infinie est rencontrée lors de la résolution du chemin, *RuntimeError* est levée.

Nouveau dans la version 3.6 : L'argument *strict*.

Path.rglob (*pattern*)

This is like calling *Path.glob()* with « **** » added in front of the given *pattern* :

```
>>> sorted(Path().rglob("*.py"))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Path.rmdir ()

Supprime ce dossier. Le dossier doit être vide.

Path.samefile (*other_path*)

Renvoie si ce chemin pointe vers le même fichier que *other_path*, qui peut être soit un chemin, soit une chaîne de caractères. La sémantique est similaire à *os.path.samefile()* et *os.path.samestat()*.

OSError peut être levée si l'un des fichiers ne peut être accédé pour quelque raison.

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

Nouveau dans la version 3.5.

Path.symlink_to (*target*, *target_is_directory=False*)

Fait de ce chemin un lien symbolique vers *target*. Sous Windows, *target_is_directory* doit être vrai (la valeur par défaut étant *False*) si la cible du lien est un dossier. Sous POSIX, la valeur de *target_is_directory* est ignorée.

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

Note : L'ordre des arguments (lien, cible) est l'opposé de ceux de *os.symlink()*.

Path.touch (*mode=0o666*, *exist_ok=True*)

Créer un fichier au chemin donné. Si *mode* est fourni, il est combiné avec la valeur de l'*umask* du processus pour déterminer le mode du fichier et les drapeaux d'accès. Si le fichier existe déjà, la fonction réussit si *exist_ok* est vrai (et si l'heure de modification est mise à jour avec l'heure courante), sinon *FileExistsError* est levée.

Path.unlink ()

Supprime ce fichier ou lien symbolique. Si le chemin pointe vers un dossier, utilisez *Path.rmdir()* à la place.

`Path.write_bytes(data)`

Ouvre le fichier pointé en mode binaire, écrit *data* dedans, et ferme le fichier :

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Le fichier du même nom, s'il existe, est écrasé.

Nouveau dans la version 3.5.

`Path.write_text(data, encoding=None, errors=None)`

Ouvre le fichier pointé en mode texte, écrit *data* dedans, et ferme le fichier :

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

Nouveau dans la version 3.5.

11.2 `os.path` — manipulation courante des chemins

Code source : [Lib/posixpath.py](#) (pour POSIX), [Lib/ntpath.py](#) (pour Windows NT), et [Lib/macpath.py](#) (pour Macintosh)

Ce module implémente certaines fonctions utiles sur le nom des chemins. Pour lire ou écrire des fichiers, voir [open\(\)](#), et pour accéder au système de fichier, voir le module [os](#). Les paramètres de chemin d'accès peuvent être passés sous forme de chaînes de caractères ou de chaîne d'octets. Les programmes sont encouragés à représenter les noms de fichiers en tant que chaînes de caractères Unicode. Malheureusement, certains noms de fichiers peuvent ne pas être représentés sous forme de chaînes de caractères sous UNIX, ainsi, les programmes qui doivent prendre en charge les noms de fichiers arbitraires sur UNIX doivent utiliser des chaînes d'octets pour représenter leurs chemins d'accès. Inversement, l'utilisation de chaîne d'octets ne peut pas représenter tous les noms de fichiers sous Windows (dans le codage `mbcs` standard), par conséquent les applications Windows doivent utiliser des chaînes de caractères Unicode pour accéder à tous les fichiers.

Contrairement à une invite de commandes Unix, Python ne fait aucune extension de chemin *automatique*. Des fonctions telles que [expanduser\(\)](#) et [expandvars\(\)](#) peuvent être appelées explicitement lorsqu'une application souhaite une extension de chemin semblable à celui d'une invite de commande (voir aussi le module [glob](#)).

Voir aussi :

Le module [pathlib](#) offre une représentation objet de haut niveau des chemins.

Note : Toutes ces fonctions n'acceptent que des chaînes d'octets ou des chaînes de caractères en tant que paramètres. Le résultat est un objet du même type si un chemin ou un nom de fichier est renvoyé.

Note : Comme les différents systèmes d'exploitation ont des conventions de noms de chemins différentes, il existe plusieurs versions de ce module dans la bibliothèque standard. Le module [os.path](#) est toujours le module de chemin adapté au système d'exploitation sur lequel Python tourne, et donc adapté pour les chemins locaux. Cependant, vous pouvez également importer et utiliser les modules individuels si vous voulez manipuler un chemin qui est *toujours* dans l'un des différents formats. Ils ont tous la même interface :

- `posixpath` pour les chemins de type UNIX

- `ntpath` pour les chemins Windows
 - `macpath` pour l'ancienne forme des chemins MacOS
-

`os.path.abspath(path)`

Renvoie une version absolue et normalisée du chemin d'accès *path*. Sur la plupart des plates-formes, cela équivaut à appeler la fonction `normpath()` comme suit : `normpath(join(os.getcwd(), chemin))``.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.basename(path)`

Renvoie le nom de base du chemin d'accès *path*. C'est le second élément de la paire renvoyée en passant *path* à la fonction `split()`. Notez que le résultat de cette fonction est différent de celui du programme Unix **basename** ; là où **basename** pour `'/foo/bar/'` renvoie `'bar'`, la fonction `basename()` renvoie une chaîne vide (`' '`).

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.commonpath(paths)`

Return the longest common sub-path of each pathname in the sequence *paths*. Raise `ValueError` if *paths* contains both absolute and relative pathnames, or if *paths* is empty. Unlike `commonprefix()`, this returns a valid path.

Disponibilité Unix, Windows

Nouveau dans la version 3.5.

Modifié dans la version 3.6 : Accepts a sequence of *path-like objects*.

`os.path.commonprefix(list)`

Return the longest path prefix (taken character-by-character) that is a prefix of all paths in *list*. If *list* is empty, return the empty string (`' '`).

Note : This function may return invalid paths because it works a character at a time. To obtain a valid path, see `commonpath()`.

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.dirname(path)`

Return the directory name of pathname *path*. This is the first element of the pair returned by passing *path* to the function `split()`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.exists(path)`

Return `True` if *path* refers to an existing path or an open file descriptor. Returns `False` for broken symbolic links. On some platforms, this function may return `False` if permission is not granted to execute `os.stat()` on the requested file, even if the *path* physically exists.

Modifié dans la version 3.3 : *path* can now be an integer : `True` is returned if it is an open file descriptor, `False` otherwise.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.lexists(path)`

Return `True` if *path* refers to an existing path. Returns `True` for broken symbolic links. Equivalent to `exists()` on platforms lacking `os.lstat()`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.expanduser (*path*)

On Unix and Windows, return the argument with an initial component of `~` or `~user` replaced by that *user*'s home directory.

On Unix, an initial `~` is replaced by the environment variable `HOME` if it is set; otherwise the current user's home directory is looked up in the password directory through the built-in module `pwd`. An initial `~user` is looked up directly in the password directory.

On Windows, `HOME` and `USERPROFILE` will be used if set, otherwise a combination of `HOMEPATH` and `HOMEDRIVE` will be used. An initial `~user` is handled by stripping the last directory component from the created user path derived above.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.expandvars (*path*)

Return the argument with environment variables expanded. Substrings of the form `$name` or `${name}` are replaced by the value of environment variable *name*. Malformed variable names and references to non-existing variables are left unchanged.

On Windows, `%name%` expansions are supported in addition to `$name` and `${name}`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.getatime (*path*)

Return the time of last access of *path*. The return value is a number giving the number of seconds since the epoch (see the `time` module). Raise `OSError` if the file does not exist or is inaccessible.

If `os.stat_float_times()` returns `True`, the result is a floating point number.

os.path.getmtime (*path*)

Return the time of last modification of *path*. The return value is a number giving the number of seconds since the epoch (see the `time` module). Raise `OSError` if the file does not exist or is inaccessible.

If `os.stat_float_times()` returns `True`, the result is a floating point number.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.getctime (*path*)

Return the system's `ctime` which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for *path*. The return value is a number giving the number of seconds since the epoch (see the `time` module). Raise `OSError` if the file does not exist or is inaccessible.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.getsize (*path*)

Return the size, in bytes, of *path*. Raise `OSError` if the file does not exist or is inaccessible.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.isabs (*path*)

Return `True` if *path* is an absolute pathname. On Unix, that means it begins with a slash, on Windows that it begins with a (back)slash after chopping off a potential drive letter.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.isfile (*path*)

Return `True` if *path* is an *existing* regular file. This follows symbolic links, so both `islink()` and `isfile()` can be true for the same path.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.isdir (*path*)

Return `True` if *path* is an *existing* directory. This follows symbolic links, so both `islink()` and `isdir()` can be true for the same path.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.islink(path)`

Return `True` if *path* refers to an *existing* directory entry that is a symbolic link. Always `False` if symbolic links are not supported by the Python runtime.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.ismount(path)`

Return `True` if pathname *path* is a *mount point* : a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*'s parent, `path/..`, is on a different device than *path*, or whether `path/..` and *path* point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants. On Windows, a drive letter root and a share UNC are always mount points, and for any other path `GetVolumePathName` is called to see if it is different from the input path.

Nouveau dans la version 3.4 : Support for detecting non-root mount points on Windows.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.join(path, *paths)`

Join one or more path components intelligently. The return value is the concatenation of *path* and any members of **paths* with exactly one directory separator (`os.sep`) following each non-empty part except the last, meaning that the result will only end in a separator if the last part is empty. If a component is an absolute path, all previous components are thrown away and joining continues from the absolute path component.

On Windows, the drive letter is not reset when an absolute path component (e.g., `r'\foo'`) is encountered. If a component contains a drive letter, all previous components are thrown away and the drive letter is reset. Note that since there is a current directory for each drive, `os.path.join("c:", "foo")` represents a path relative to the current directory on drive C : (`c:foo`), not `c:\foo`.

Modifié dans la version 3.6 : Accepts a *path-like object* for *path* and *paths*.

`os.path.normcase(path)`

Normalize the case of a pathname. On Unix and Mac OS X, this returns the path unchanged ; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes. Raise a `TypeError` if the type of *path* is not `str` or `bytes` (directly or indirectly through the *os.PathLike* interface).

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.normpath(path)`

Normalize a pathname by collapsing redundant separators and up-level references so that `A//B`, `A/B/`, `A/./B` and `A/foo/./B` all become `A/B`. This string manipulation may change the meaning of a path that contains symbolic links. On Windows, it converts forward slashes to backward slashes. To normalize case, use *normcase()*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.realpath(path)`

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system).

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.relpath(path, start=os.curdir)`

Return a relative filepath to *path* either from the current directory or from an optional *start* directory. This is a path computation : the filesystem is not accessed to confirm the existence or nature of *path* or *start*.

start defaults to *os.curdir*.

Disponibilité : Unix, Windows.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.samefile(path1, path2)`

Return `True` if both pathname arguments refer to the same file or directory. This is determined by the device number and i-node number and raises an exception if an *os.stat()* call on either pathname fails.

Disponibilité : Unix, Windows.

Modifié dans la version 3.2 : Prise en charge de Windows.

Modifié dans la version 3.4 : Windows now uses the same implementation as all other platforms.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.sameopenfile(fp1, fp2)`

Return True if the file descriptors *fp1* and *fp2* refer to the same file.

Disponibilité : Unix, Windows.

Modifié dans la version 3.2 : Prise en charge de Windows.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.samestat(stat1, stat2)`

Return True if the stat tuples *stat1* and *stat2* refer to the same file. These structures may have been returned by `os.fstat()`, `os.lstat()`, or `os.stat()`. This function implements the underlying comparison used by `samefile()` and `sameopenfile()`.

Disponibilité : Unix, Windows.

Modifié dans la version 3.4 : Prise en charge de Windows.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.split(path)`

Split the pathname *path* into a pair, (*head*, *tail*) where *tail* is the last pathname component and *head* is everything leading up to that. The *tail* part will never contain a slash; if *path* ends in a slash, *tail* will be empty. If there is no slash in *path*, *head* will be empty. If *path* is empty, both *head* and *tail* are empty. Trailing slashes are stripped from *head* unless it is the root (one or more slashes only). In all cases, `join(head, tail)` returns a path to the same location as *path* (but the strings may differ). Also see the functions `dirname()` and `basename()`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.splitdrive(path)`

Split the pathname *path* into a pair (*drive*, *tail*) where *drive* is either a mount point or the empty string. On systems which do not use drive specifications, *drive* will always be the empty string. In all cases, *drive* + *tail* will be the same as *path*.

On Windows, splits a pathname into drive/UNC sharepoint and relative path.

If the path contains a drive letter, *drive* will contain everything up to and including the colon. e.g. `splitdrive("c:/dir")` returns ("c:", "/dir")

If the path contains a UNC path, *drive* will contain the host name and share, up to but not including the fourth separator. e.g. `splitdrive("//host/computer/dir")` returns ("//host/computer", "/dir")

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.splitext(path)`

Split the pathname *path* into a pair (*root*, *ext*) such that *root* + *ext* == *path*, and *ext* is empty or begins with a period and contains at most one period. Leading periods on the basename are ignored; `splitext('.cshrc')` returns ('.cshrc', '').

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.splitunc(path)`

Obsolète depuis la version 3.1 : Use `splitdrive` instead.

Split the pathname *path* into a pair (*unc*, *rest*) so that *unc* is the UNC mount point (such as `r'\\host\mount'`), if present, and *rest* the rest of the path (such as `r'path\file.ext'`). For paths containing drive letters, *unc* will always be the empty string.

Disponibilité : Windows.

`os.path.supports_unicode_filenames`

True if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system).

11.3 `fileinput` — Iterate over lines from multiple input streams

Code source : [Lib/fileinput.py](#)

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see [open\(\)](#).

The typical use is :

```
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in `sys.argv[1:]`, defaulting to `sys.stdin` if the list is empty. If a filename is '-', it is also replaced by `sys.stdin`. To specify an alternative list of filenames, pass it as the first argument to `input()`. A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the *mode* parameter in the call to `input()` or `FileInput`. If an I/O error occurs during opening or reading a file, *OSError* is raised.

Modifié dans la version 3.3 : *IOError* used to be raised; it is now an alias of *OSError*.

If `sys.stdin` is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using `sys.stdin.seek(0)`).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the *openhook* parameter to `fileinput.input()` or `FileInput`. The hook must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. Two useful hooks are already provided by this module.

The following function is the primary interface of this module :

`fileinput.input(files=None, inplace=False, backup="", bufsize=0, mode='r', openhook=None)`

Create an instance of the `FileInput` class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the `FileInput` class.

The `FileInput` instance can be used as a context manager in the `with` statement. In this example, `input` is closed after the `with` statement is exited, even if an exception occurs :

```
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

Modifié dans la version 3.2 : Can be used as a context manager.

Deprecated since version 3.6, will be removed in version 3.8 : The *bufsize* parameter.

The following functions use the global state created by `fileinput.input()`; if there is no active state, *RuntimeError* is raised.

`fileinput.filename()`

Return the name of the file currently being read. Before the first line has been read, returns `None`.

`fileinput.fileno()`

Return the integer « file descriptor » for the current file. When no file is opened (before the first line and between files), returns `-1`.

`fileinput.lineno()`

Return the cumulative line number of the line that has just been read. Before the first line has been read, returns 0. After the last line of the last file has been read, returns the line number of that line.

`fileinput.filelineno()`

Return the line number in the current file. Before the first line has been read, returns 0. After the last line of the last file has been read, returns the line number of that line within the file.

`fileinput.isfirstline()`

Returns true if the line just read is the first line of its file, otherwise returns false.

`fileinput.isstdin()`

Returns true if the last line was read from `sys.stdin`, otherwise returns false.

`fileinput.nextfile()`

Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

`fileinput.close()`

Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well :

class `fileinput.FileInput` (*files=None*, *inplace=False*, *backup=""*, *bufsize=0*, *mode='r'*, *openhook=None*)

Class `FileInput` is the implementation; its methods `filename()`, `fileno()`, `lineno()`, `filelineno()`, `isfirstline()`, `isstdin()`, `nextfile()` and `close()` correspond to the functions of the same name in the module. In addition it has a `readline()` method which returns the next input line, and a `__getitem__()` method which implements the sequence behavior. The sequence must be accessed in strictly sequential order; random access and `readline()` cannot be mixed.

With *mode* you can specify which file mode will be passed to `open()`. It must be one of 'r', 'rU', 'U' and 'rb'.

The *openhook*, when given, must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. You cannot use *inplace* and *openhook* together.

A `FileInput` instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs :

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

Modifié dans la version 3.2 : Can be used as a context manager.

Obsolète depuis la version 3.4 : The 'rU' and 'U' modes.

Deprecated since version 3.6, will be removed in version 3.8 : The *bufsize* parameter.

Optional in-place filtering : if the keyword argument *inplace=True* is passed to `fileinput.input()` or to the `FileInput` constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the *backup* parameter is given (typically as *backup='.<some extension>'*), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is '.bak' and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

The two following opening hooks are provided by this module :

`fileinput.hook_compressed` (*filename*, *mode*)

Transparently opens files compressed with gzip and bzip2 (recognized by the extensions '.gz' and '.bz2') using the `gzip` and `bz2` modules. If the filename extension is not '.gz' or '.bz2', the file is opened normally (ie, using `open()` without any decompression).

Usage example : `fi = fileinput.FileInput(openhook=fileinput.hook_compressed)`
`fileinput.hook_encoded(encoding, errors=None)`
Returns a hook which opens each file with `open()`, using the given *encoding* and *errors* to read the file.
Usage example : `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`
Modifié dans la version 3.6 : Added the optional *errors* parameter.

11.4 stat — Interpreting stat() results

Source code : [Lib/stat.py](#)

The `stat` module defines constants and functions for interpreting the results of `os.stat()`, `os.fstat()` and `os.lstat()` (if they exist). For complete details about the `stat()`, `fstat()` and `lstat()` calls, consult the documentation for your system.

Modifié dans la version 3.4 : The `stat` module is backed by a C implementation.

The `stat` module defines the following functions to test for specific file types :

`stat.S_ISDIR(mode)`
Return non-zero if the mode is from a directory.

`stat.S_ISCHR(mode)`
Return non-zero if the mode is from a character special device file.

`stat.S_ISBLK(mode)`
Return non-zero if the mode is from a block special device file.

`stat.S_ISREG(mode)`
Return non-zero if the mode is from a regular file.

`stat.S_ISFIFO(mode)`
Return non-zero if the mode is from a FIFO (named pipe).

`stat.S_ISLNK(mode)`
Return non-zero if the mode is from a symbolic link.

`stat.S_ISSOCK(mode)`
Return non-zero if the mode is from a socket.

`stat.S_ISDOOR(mode)`
Return non-zero if the mode is from a door.
Nouveau dans la version 3.4.

`stat.S_ISPORT(mode)`
Return non-zero if the mode is from an event port.
Nouveau dans la version 3.4.

`stat.S_ISWHT(mode)`
Return non-zero if the mode is from a whiteout.
Nouveau dans la version 3.4.

Two additional functions are defined for more general manipulation of the file's mode :

`stat.S_IMODE(mode)`
Return the portion of the file's mode that can be set by `os.chmod()`—that is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).

`stat.S_IFMT(mode)`

Return the portion of the file’s mode that describes the file type (used by the `S_IS*` () functions above).

Normally, you would use the `os.path.is*` () functions for testing the type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the `stat()` system call for each test. These are also useful when checking for information about a file that isn’t handled by `os.path`, like the tests for block and character devices.

Exemple :

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
    calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

An additional utility function is provided to convert a file’s mode in a human readable string :

`stat.filemode(mode)`

Convert a file’s mode to a string of the form “-rwxrwxrwx”.

Nouveau dans la version 3.3.

Modifié dans la version 3.4 : The function supports `S_IFDOOR`, `S_IFPORT` and `S_IFWHT`.

All the variables below are simply symbolic indexes into the 10-tuple returned by `os.stat()`, `os.fstat()` or `os.lstat()`.

`stat.ST_MODE`

Inode protection mode.

`stat.ST_INO`

Numéro d’inode.

`stat.ST_DEV`

Device inode resides on.

`stat.ST_NLINK`

Number of links to the inode.

`stat.ST_UID`

User id of the owner.

`stat.ST_GID`

Group id of the owner.

`stat.ST_SIZE`

Size in bytes of a plain file ; amount of data waiting on some special files.

`stat.ST_ATIME`

Time of last access.

`stat.ST_MTIME`

Time of last modification.

`stat.ST_CTIME`

The « ctime » as reported by the operating system. On some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of « file size » changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the « size » is the number of bytes waiting to be read at the time of the call to `os.stat()`, `os.fstat()`, or `os.lstat()` ; this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the `ST_MODE` field.

Use of the functions above is more portable than use of the first set of flags :

`stat.S_IFSOCK`

Socket.

`stat.S_IFLNK`

Symbolic link.

`stat.S_IFREG`

Regular file.

`stat.S_IFBLK`

Block device.

`stat.S_IFDIR`

Dossier.

`stat.S_IFCHR`

Character device.

`stat.S_IFIFO`

FIFO.

`stat.S_IFDOOR`

Door.

Nouveau dans la version 3.4.

`stat.S_IFPORT`

Event port.

Nouveau dans la version 3.4.

`stat.S_IFWHT`

Whiteout.

Nouveau dans la version 3.4.

Note : `S_IFDOOR`, `S_IFPORT` or `S_IFWHT` are defined as 0 when the platform does not have support for the file types.

The following flags can also be used in the *mode* argument of `os.chmod()` :

`stat.S_ISUID`
Set UID bit.

`stat.S_ISGID`
Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory : files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the `S_ISGID` bit set. For a file that does not have the group execution bit (`S_IXGRP`) set, the set-group-ID bit indicates mandatory file/record locking (see also `S_ENFMT`).

`stat.S_ISVTX`
Sticky bit. When this bit is set on a directory it means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, or by a privileged process.

`stat.S_IRWXU`
Mask for file owner permissions.

`stat.S_IRUSR`
Owner has read permission.

`stat.S_IWUSR`
Owner has write permission.

`stat.S_IXUSR`
Owner has execute permission.

`stat.S_IRWXG`
Mask for group permissions.

`stat.S_IRGRP`
Group has read permission.

`stat.S_IWGRP`
Group has write permission.

`stat.S_IXGRP`
Group has execute permission.

`stat.S_IRWXO`
Mask for permissions for others (not in group).

`stat.S_IROTH`
Others have read permission.

`stat.S_IWOTH`
Others have write permission.

`stat.S_IXOTH`
Others have execute permission.

`stat.S_ENFMT`
System V file locking enforcement. This flag is shared with `S_ISGID` : file/record locking is enforced on files that do not have the group execution bit (`S_IXGRP`) set.

`stat.S_IREAD`
Unix V7 synonym for `S_IRUSR`.

`stat.S_IWRITE`
Unix V7 synonym for `S_IWUSR`.

`stat.S_IXEXEC`

Unix V7 synonym for `S_IXUSR`.

The following flags can be used in the *flags* argument of `os.chflags()` :

`stat.UF_NODUMP`

Do not dump the file.

`stat.UF_IMMUTABLE`

The file may not be changed.

`stat.UF_APPEND`

The file may only be appended to.

`stat.UF_OPAQUE`

The directory is opaque when viewed through a union stack.

`stat.UF_NOUNLINK`

The file may not be renamed or deleted.

`stat.UF_COMPRESSED`

The file is stored compressed (Mac OS X 10.6+).

`stat.UF_HIDDEN`

The file should not be displayed in a GUI (Mac OS X 10.5+).

`stat.SF_ARCHIVED`

The file may be archived.

`stat.SF_IMMUTABLE`

The file may not be changed.

`stat.SF_APPEND`

The file may only be appended to.

`stat.SF_NOUNLINK`

The file may not be renamed or deleted.

`stat.SF_SNAPSHOT`

The file is a snapshot file.

See the *BSD or Mac OS systems man page `chflags(2)` for more information.

On Windows, the following file attribute constants are available for use when testing bits in the `st_file_attributes` member returned by `os.stat()`. See the [Windows API documentation](#) for more detail on the meaning of these constants.

`stat.FILE_ATTRIBUTE_ARCHIVE`

`stat.FILE_ATTRIBUTE_COMPRESSED`

`stat.FILE_ATTRIBUTE_DEVICE`

`stat.FILE_ATTRIBUTE_DIRECTORY`

`stat.FILE_ATTRIBUTE_ENCRYPTED`

`stat.FILE_ATTRIBUTE_HIDDEN`

`stat.FILE_ATTRIBUTE_INTEGRITY_STREAM`

`stat.FILE_ATTRIBUTE_NORMAL`

`stat.FILE_ATTRIBUTE_NOT_CONTENT_INDEXED`

`stat.FILE_ATTRIBUTE_NO_SCRUB_DATA`

`stat.FILE_ATTRIBUTE_OFFLINE`

`stat.FILE_ATTRIBUTE_READONLY`

`stat.FILE_ATTRIBUTE_REPARSE_POINT`

`stat.FILE_ATTRIBUTE_SPARSE_FILE`

`stat.FILE_ATTRIBUTE_SYSTEM`

`stat.FILE_ATTRIBUTE_TEMPORARY`
`stat.FILE_ATTRIBUTE_VIRTUAL`
 Nouveau dans la version 3.5.

11.5 filecmp – Comparaisons de fichiers et de répertoires

Code source : [Lib/filecmp.py](#)

Le module `filecmp` définit les fonctions permettant de comparer les fichiers et les répertoires, avec différents compromis optionnels durée / exactitude. Pour comparer des fichiers, voir aussi le module `difflib`.

Le module `filecmp` définit les fonctions suivantes :

`filecmp.cmp(f1, f2, shallow=True)`

Compare les fichiers nommés *f1* et *f2*, renvoie `True` s'ils semblent égaux, `False` sinon.

Si *shallow* est vrai, les fichiers avec des signatures `os.stat()` identiques sont considérés comme égaux. Sinon, le contenu des fichiers est comparé.

Notez qu'aucun programme externe n'est appelé à partir de cette fonction, ce qui lui confère des qualités de portabilité et d'efficacité.

Cette fonction utilise un cache pour les comparaisons antérieures et les résultats, les entrées du cache étant invalidées si les informations `os.stat()` du fichier sont modifiées. La totalité du cache peut être effacée avec `clear_cache()`.

`filecmp.cmpfiles(dir1, dir2, common, shallow=True)`

Compare les fichiers des deux répertoires *dir1* et *dir2* dont les noms sont donnés par *common*.

Renvoie trois listes de noms de fichiers : *match*, *mismatch*, *errors*. *match* contient la liste des fichiers qui correspondent, *mismatch* contient les noms de ceux qui ne correspondent pas et *errors* répertorie les noms des fichiers qui n'ont pas pu être comparés. Les fichiers sont répertoriés dans *errors* s'ils n'existent pas dans l'un des répertoires, si l'utilisateur ne dispose pas de l'autorisation nécessaire pour les lire ou si la comparaison n'a pas pu être effectuée pour une autre raison.

Le paramètre *shallow* a la même signification et la même valeur par défaut que pour `filecmp.cmp()`.

Par exemple, `cmpfiles('a', 'b', ['c', 'd/e'])` compare *a/c* et *b/c* et *a/d/e* avec *b/d/e*. 'c' et 'd/e' seront chacun dans l'une des trois listes renvoyées.

`filecmp.clear_cache()`

Efface le cache `filecmp`. Cela peut être utile si un fichier est comparé juste après avoir été modifié (dans un délai inférieur à la résolution *mtime* du système de fichiers sous-jacent).

Nouveau dans la version 3.4.

11.5.1 La classe `dircmp`

`class filecmp.dircmp(a, b, ignore=None, hide=None)`

Construit un nouvel objet de comparaison de répertoires, pour comparer les répertoires *a* et *b*. *ignore* est une liste de noms à ignorer, par défaut à `filecmp.DEFAULT_IGNORES`. *hide* est une liste de noms à cacher, par défaut à `[os.curdir, os.pardir]`.

La classe `dircmp` compare les fichiers en faisant des comparaisons *superficielles* comme décrit pour `filecmp.cmp()`.

La classe `dircmp` fournit les méthodes suivantes :

`report()`

Affiche (sur `sys.stdout`) une comparaison entre *a* et *b*.

report_partial_closure()

Affiche une comparaison entre *a* et *b* et les sous-répertoires immédiats communs.

report_full_closure()

Affiche une comparaison entre *a* et *b* et les sous-répertoires communs (récursivement).

La classe `dircmp` offre un certain nombre d'attributs intéressants qui peuvent être utilisés pour obtenir diverses informations sur les arborescences de répertoires comparées.

Notez que, via les points d'ancrage `__getattr__()`, tous les attributs sont calculés de manière paresseuse. Il n'y a donc pas de pénalité en vitesse si seuls les attributs rapides à calculer sont utilisés.

left

Le répertoire *a*.

right

Le répertoire *b*.

left_list

Fichiers et sous-répertoires dans *a*, filtrés par *hide* et *ignore*.

right_list

Fichiers et sous-répertoires dans *b*, filtrés par *hide* et *ignore*.

common

Fichiers et sous-répertoires à la fois dans *a* et *b*.

left_only

Fichiers et sous-répertoires uniquement dans *a*.

right_only

Fichiers et sous-répertoires uniquement dans *b*.

common_dirs

Sous-répertoires à la fois dans *a* et *b*.

common_files

Fichiers à la fois dans *a* et *b*.

common_funny

Noms dans *a* et *b*, tels que le type diffère entre les répertoires, ou noms pour lesquels `os.stat()` signale une erreur.

same_files

Fichiers identiques dans *a* et *b*, en utilisant l'opérateur de comparaison de fichiers de la classe.

diff_files

Fichiers figurant à la fois dans *a* et dans *b*, dont le contenu diffère en fonction de l'opérateur de comparaison de fichiers de la classe.

funny_files

Fichiers à la fois dans *a* et dans *b*, mais ne pouvant pas être comparés.

subdirs

Un dictionnaire faisant correspondre les noms dans `common_dirs` vers des objets `dircmp`.

`filecmp.DEFAULT_IGNORES`

Nouveau dans la version 3.4.

Liste des répertoires ignorés par défaut par `dircmp`.

Voici un exemple simplifié d'utilisation de l'attribut `subdirs` pour effectuer une recherche récursive dans deux répertoires afin d'afficher des fichiers communs différents :

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...             dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
```

(suite sur la page suivante)

(suite de la page précédente)

```

...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)

```

11.6 tempfile — Génération de fichiers et répertoires temporaires

Code source : [Lib/tempfile.py](#)

Ce module crée des fichiers et répertoires temporaires. Il fonctionne sur toutes les plateformes supportées. *TemporaryFile*, *NamedTemporaryFile*, *TemporaryDirectory*, et *SpooledTemporaryFile* sont des interfaces haut-niveau qui fournissent un nettoyage automatique et peuvent être utilisées comme gestionnaire de contexte. *mkstemp()* et *mkdtemp()* sont des fonctions bas-niveau qui nécessitent un nettoyage manuel.

Toutes les fonctions et constructeurs appelables par l'utilisateur ont des arguments additionnels qui permettent de contrôler directement le chemin et le nom des répertoires et fichiers. Les noms de fichiers utilisés par ce module incluent une chaîne de caractères aléatoires qui leur permet d'être créés de manière sécurisée dans des répertoires temporaires partagés. Afin de maintenir la compatibilité descendante, l'ordre des arguments est quelque peu étrange ; pour des questions de clarté, il est recommandé d'utiliser les arguments nommés.

Le module définit les éléments suivants pouvant être appelés par l'utilisateur :

`tempfile.TemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None)`

Renvoie un *objet fichier* qui peut être utilisé comme une zone de stockage temporaire. Le fichier est créé de manière sécurisée, utilisant les mêmes règles que *mkstemp()*. Il sera détruit dès qu'il sera fermé (y compris lorsque le fichier est implicitement fermé quand il est collecté par le ramasse-miette). Sous Unix, l'entrée du répertoire n'est soit pas du tout créée, ou est supprimée immédiatement après sa création. Les autres plateformes ne gèrent pas cela, votre code ne doit pas compter sur un fichier temporaire créé en utilisant cette fonction ayant ou non un nom visible sur le système de fichier.

L'objet résultat peut être utilisé comme un gestionnaire de contexte (voir *Exemples*). Une fois le contexte ou la destruction de l'objet fichier terminé, le fichier temporaire sera supprimé du système de fichiers.

Le paramètre *mode* vaut par défaut `'w+b'` afin que le fichier créé puisse être lu et écrit sans être fermé. Le mode binaire est utilisé afin que le comportement soit le même sur toutes les plateformes quelque soit la donnée qui est stockée. *buffering*, *encoding* et *newline* sont interprétés de la même façon que pour *open()*.

Les paramètres *dir*, *prefix* et *suffix* ont la même signification et même valeur par défaut que *mkstemp()*.

L'objet renvoyé est un véritable fichier sur les plateformes POSIX. Sur les autres plateformes, un objet fichier-compatible est retourné où l'attribut *file* est le véritable fichier.

L'option `os.O_TMPFILE` est utilisé s'il est disponible et fonctionne (Linux exclusivement, nécessite un noyau Linux 3.11 ou plus).

Modifié dans la version 3.5 : L'option `os.O_TMPFILE` est maintenant utilisé si disponible.

`tempfile.NamedTemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None, delete=True)`

Cette fonction fonctionne exactement comme *TemporaryFile()*, à la différence qu'il est garanti que le fichier soit visible dans le système de fichier (sur Unix, l'entrée du répertoire est supprimée). Le nom peut être récupéré depuis l'attribut *name* de l'objet fichier-compatible retourné. Le fait que le nom puisse être utilisé pour ouvrir le fichier une seconde fois, tant que le fichier temporaire nommé est toujours ouvert, varie entre les plateformes (cela peut l'être sur Unix, mais c'est impossible sur Windows NT et plus). Si *delete* est vrai (valeur par défaut), le fichier est supprimé dès qu'il est fermé. L'objet retourné est toujours un objet fichier-compatible où l'attribut *file* est le véritable fichier. L'objet fichier-compatible peut être utilisé dans un gestionnaire de contexte (instruction *with*), juste comme un fichier normal.

`tempfile.SpooledTemporaryFile` (*max_size=0, mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None*)

Cette fonction se comporte exactement comme `TemporaryFile()`, à l'exception que les données sont stockées en mémoire jusqu'à ce que leur taille dépasse *max_size*, ou que la méthode `fileno()` soit appelée. À ce moment, le contenu est écrit sur disque et le fonctionnement redevient similaire à celui de `TemporaryFile()`.

Le fichier renvoyé a une méthode supplémentaire, `rollover()`, qui provoque la mise en écriture sur disque quelque soit la taille du fichier.

L'objet renvoyé est un objet fichier-compatible où l'attribut `_file` est soit un objet `io.BytesIO` soit un objet `io.StringIO` (en fonction du *mode*) soit un véritable fichier, si la fonction `rollover()` a été appelée. Cet objet fichier-compatible peut être utilisé dans un gestionnaire de contexte (instruction `with`), comme un fichier normal.

Modifié dans la version 3.3 : la méthode de troncature accepte maintenant un argument `size`.

`tempfile.TemporaryDirectory` (*suffix=None, prefix=None, dir=None*)

Cette fonction crée un répertoire temporaire de manière sécurisée utilisant les mêmes règles que `mkdtemp()`. L'objet renvoyé peut être utilisé comme un gestionnaire de contexte (voir *Exemples*). À la sortie du contexte d'exécution ou à la destruction de l'objet, le répertoire temporaire et tout son contenu sont supprimés du système de fichiers.

The directory name can be retrieved from the `name` attribute of the returned object. When the returned object is used as a context manager, the `name` will be assigned to the target of the `as` clause in the `with` statement, if there is one.

Le répertoire peut être explicitement nettoyé en appelant la méthode `cleanup()`.

Nouveau dans la version 3.2.

`tempfile.mkstemp` (*suffix=None, prefix=None, dir=None, text=False*)

Crée un fichier temporaire de la manière la plus sécurisée qui soit. Il n'y a pas d'accès concurrent (*race condition*) au moment de la création du fichier, en supposant que la plateforme implémente correctement l'option `os.O_EXCL` pour `os.open()`. Le fichier est seulement accessible en lecture et écriture par l'ID de l'utilisateur créateur. Si la plateforme utilise des bits de permissions pour indiquer si le fichier est exécutable, alors le fichier n'est exécutable par personne. Le descripteur de fichier n'est pas hérité par les processus fils.

À la différence de `TemporaryFile()`, l'utilisateur de `mkstemp()` est responsable de la suppression du fichier temporaire quand il n'en a plus besoin.

Si *suffix* ne vaut pas `None`, le nom de fichier se terminera avec ce suffixe, sinon il n'y aura pas de suffixe. `mkstemp()` ne met pas de point entre le nom du fichier et le suffixe. Si vous en avez besoin, mettez le point au début de *suffix*.

Si *prefix* ne vaut pas `None`, le nom de fichier commencera avec ce préfixe, sinon un préfixe par défaut est utilisé. La valeur par défaut est la valeur retournée par `gettempprefix()` ou `gettempprefixb()`.

Si *dir* ne vaut pas `None`, le fichier sera créé dans ce répertoire, autrement, un répertoire par défaut sera utilisé. Le répertoire par défaut est choisi depuis une liste dépendante de la plateforme, mais l'utilisateur de l'application peut contrôler l'emplacement du répertoire en spécifiant les variables d'environnement `TMPDIR`, `TEMP` ou `TMP`. Il n'y a pas de garantie que le nom de fichier généré aura de bonnes propriétés telles que ne pas avoir besoin de le mettre entre guillemets lorsque celui-ci est passé à des commandes externes via `os.popen()`.

Si l'un des paramètres *suffix*, *prefix* et *dir* n'est pas `None`, ils doivent être du même type. S'ils sont de type `bytes`, le nom renvoyé sera de type `bytes` plutôt que de type `str`. Si vous voulez forcer la valeur renvoyée en `bytes`, passez `suffix=b''`.

Si *text* est spécifié, cela indique si le fichier doit être ouvert en mode binaire (par défaut) ou en mode texte. Sur certaines plateformes, cela ne fait aucune différence.

`mkstemp()` renvoie un n-uplet contenant un descripteur (*handle* en anglais) au niveau du système d'exploitation vers un fichier ouvert (le même que renvoie `os.open()`) et le chemin d'accès absolu de ce fichier, dans cet ordre. Modifié dans la version 3.5 : *suffix*, *prefix*, et *dir* peuvent maintenant être spécifiés en `bytes` pour obtenir un résultat en `bytes`. Avant cela, le type `str` était le seul autorisé. *suffix* et *prefix* acceptent maintenant la valeur par défaut `None` pour que la valeur par défaut appropriée soit utilisée.

`tempfile.mkdtemp(suffix=None, prefix=None, dir=None)`

Crée un répertoire temporaire de la manière la plus sécurisée qu'il soit. Il n'y a pas d'accès concurrent (`race condition`) au moment de la création du répertoire. Le répertoire est accessible en lecture, en écriture, et son contenu lisible uniquement pour l'ID de l'utilisateur créateur.

L'utilisateur de `mkdtemp()` est responsable de la suppression du répertoire temporaire et de son contenu lorsqu'il n'en a plus besoin.

Les arguments `prefix`, `suffix`, et `dir` sont les mêmes que pour `mkstemp()`.

`mkdtemp()` renvoie le chemin absolu du nouveau répertoire.

Modifié dans la version 3.5 : `suffix`, `prefix`, et `dir` peuvent maintenant être spécifiés en `bytes` pour obtenir un résultat en `bytes`. Avant cela, le type `str` était le seul autorisé. `suffix` et `prefix` acceptent maintenant la valeur par défaut `None` pour que la valeur par défaut appropriée soit utilisée.

`tempfile.gettempdir()`

Renvoie le nom du répertoire utilisé pour les fichiers temporaires. C'est la valeur par défaut pour l'argument `dir` de toutes les fonctions de ce module.

Python cherche un répertoire parmi une liste standard de répertoires dans lequel l'utilisateur final peut créer des fichiers. La liste est :

1. Le répertoire correspondant à la variable d'environnement `TMPDIR`.
2. Le répertoire correspondant à la variable d'environnement `TEMP`.
3. Le répertoire correspondant à la variable d'environnement `TMP`.
4. Un emplacement dépendant à la plateforme :
 - Sur Windows, les répertoires `C:\TEMP`, `C:\TMP`, `\TEMP`, et `\TMP`, dans cet ordre.
 - Sur toutes les autres plate-formes, les répertoires `/tmp`, `/var/tmp`, et `/usr/tmp`, dans cet ordre.
5. En dernier ressort, le répertoire de travail courant.

Le résultat de cette recherche est mis en cache, voir la description de `tempdir` dessous.

`tempfile.gettempdirb()`

Similaire à `gettempdir()` mais la valeur retournée est en `bytes`.

Nouveau dans la version 3.5.

`tempfile.gettempprefix()`

Renvoie le préfixe de nom de fichier utilisé pour créer les fichiers temporaires. Cela ne contient pas le nom du répertoire.

`tempfile.gettempprefixb()`

Similaire à `gettempprefix()` mais la valeur retournée est en `bytes`.

Nouveau dans la version 3.5.

Le module utilise une variable globale pour stocker le nom du répertoire utilisé pour les fichiers temporaires renvoyés par `gettempdir()`. Vous pouvez directement utiliser la variable globale pour surcharger le processus de sélection, mais ceci est déconseillé. Toutes les fonctions de ce module prennent un argument `dir` qui peut être utilisé pour spécifier le répertoire. Il s'agit de la méthode recommandée.

`tempfile.tempdir`

Quand une valeur autre que `None` est spécifiée, cette variable définit la valeur par défaut pour l'argument `dir` des fonctions définies dans ce module.

If `tempdir` is `None` (the default) at any call to any of the above functions except `gettempprefix()` it is initialized following the algorithm described in `gettempdir()`.

11.6.1 Exemples

Voici quelques exemples classiques d'utilisation du module `tempfile` :

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2 Fonctions et variables obsolètes

Historiquement, la méthode pour créer des fichiers temporaires consistait à générer un nom de fichier avec la fonction `mktemp()` puis créer un fichier en utilisant ce nom. Malheureusement, cette méthode n'est pas fiable car un autre processus peut créer un fichier avec ce nom entre l'appel à la fonction `mktemp()` et la tentative de création de fichier par le premier processus en cours. La solution est de combiner les deux étapes et de créer le fichier immédiatement. Cette approche est utilisée par `mkstemp()` et les autres fonctions décrites plus haut.

`tempfile.mktemp(suffix='', prefix='tmp', dir=None)`

Obsolète depuis la version 2.3 : Utilisez `mkstemp()` à la place.

Renvoie le chemin absolu d'un fichier qui n'existe pas lorsque l'appel est fait. Les arguments *prefix*, *suffix*, et *dir* sont similaires à ceux de `mkstemp()` mais les noms de fichiers en *bytes*, *sufix=None* et *prefix=None* ne sont pas implémentées.

Avertissement : Utiliser cette fonction peut introduire une faille de sécurité dans votre programme. Avant que vous n'ayez le temps de faire quoi que ce soit avec le nom de fichier renvoyé, quelqu'un peut l'utiliser. L'utilisation de `mktemp()` peut être remplacée facilement avec `NamedTemporaryFile()` en y passant le paramètre `delete=False` :

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmpjtjujtt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
```

```
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.7 glob — Recherche de chemins de style Unix selon certains motifs

Code source : [Lib/glob.py](#)

Le module `glob` recherche tous les chemins correspondant à un motif particulier selon les règles utilisées par le shell Unix, les résultats sont renvoyés dans un ordre arbitraire. Aucun remplacement du tilde n'est réalisé, mais les caractères `*`, `?`, et les caractères `[]` exprimant un intervalle sont correctement renvoyés. Cette opération est réalisée en utilisant les fonctions `os.scandir()` et `fnmatch.fnmatch()` de concert, et sans invoquer une sous-commande. Notons qu'à la différence de `fnmatch.fnmatch()`, `glob` traite les noms de fichiers commençant par un point (`.`) comme des cas spéciaux. (Pour remplacer le tilde et les variables shell, nous vous conseillons d'utiliser les fonctions `os.path.expanduser()` et `os.path.expandvars()`.)

Pour une correspondance littérale, il faut entourer le métacaractère par des crochets. Par exemple, `'[?]'` reconnaît le caractère `'?'`.

Voir aussi :

Le module `pathlib` offre une représentation objet de haut niveau des chemins.

`glob.glob(pathname, *, recursive=False)`

Renvoie une liste, potentiellement vide, de chemins correspondant au motif `pathname`, qui doit être une chaîne de caractères contenant la spécification du chemin. `pathname` peut être soit absolu (comme `/usr/src/Python-1.5/Makefile`) soit relatif (comme `../Tools/*/*.gif`), et contenir un caractère de remplacement de style shell. Les liens symboliques cassés sont aussi inclus dans les résultats (comme pour le shell).

Si `recursive` est vrai, le motif « `**` » reconnaît tous les fichiers et, zéro ou plus répertoires et sous-répertoires. Si le motif est suivi par un caractère de séparation `os.sep`, seuls les répertoires et sous-répertoires sont reconnus.

Note : Utiliser le motif « `**` » dans de grandes arborescences de dossier peut consommer une quantité de temps démesurée.

Modifié dans la version 3.5 : Gestion des chemins récursifs utilisant le motif « `**` ».

`glob.iglob(pathname, *, recursive=False)`

Renvoie un *iterator* qui produit les mêmes valeurs que `glob()`, sans toutes les charger en mémoire simultanément.

`glob.escape(pathname)`

Échappe tous les caractères spéciaux (`'?'`, `'*'` et `'['`). Cela est utile pour reconnaître une chaîne de caractère littérale arbitraire qui contiendrait ce type de caractères. Les caractères spéciaux dans les disques et répertoires partagés (chemins UNC) ne sont pas échappés, e.g. sous Windows `escape('///?/c:/Quo vadis?.txt')` renvoie `'///?/c:/Quo vadis[?].txt'`.

Nouveau dans la version 3.4.

Par exemple, considérons un répertoire contenant les fichiers suivants : `1.gif`, `2.txt`, `card.gif` et un sous-répertoire `sub` contenant seulement le fichier `3.txt`. `glob()` produit les résultats suivants. Notons que les composantes principales des chemins sont préservées.


```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

Si le répertoire contient des fichiers commençant par `.`, ils ne sont pas reconnus par défaut. Par exemple, considérons un répertoire contenant `card.gif` et `.card.gif` :

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.*')
['.card.gif']
```

Voir aussi :

Module `fnmatch` Recherche de noms de fichiers de style shell (ne concerne pas les chemins)

11.8 `fnmatch` — Filtrage par motif des noms de fichiers Unix

Code source : [Lib/fnmatch.py](#)

Ce module fournit la gestion des caractères de remplacement de style shell Unix, qui ne sont *pas* identiques à ceux utilisés dans les expressions régulières (documentés dans le module `re`). Les caractères spéciaux utilisés comme caractères de remplacement de style shell sont :

Motif	Signification
<code>*</code>	reconnaît n'importe quoi
<code>?</code>	reconnaît n'importe quel caractère unique
<code>[seq]</code>	reconnaît n'importe quel caractère dans <i>seq</i>
<code>[!seq]</code>	reconnaît n'importe quel caractère qui n'est pas dans <i>seq</i>

Pour une correspondance littérale, il faut entourer le métacaractère par des crochets. Par exemple, `'[?]` reconnaît le caractère `' ? '`.

Note that the filename separator (`'/'` on Unix) is *not* special to this module. See module `glob` for pathname expansion (`glob` uses `filter()` to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the `*` and `?` patterns.

`fnmatch.fnmatch(filename, pattern)`

Teste si la chaîne de caractères *filename* correspond au motif *pattern*, en renvoyant `True` ou `False`. La casse de chacun des paramètres peut être normalisée en utilisant `os.path.normcase()`. `fnmatchcase()` peut être utilisée pour réaliser une comparaison sensible à la casse, indépendamment du système d'exploitation.

Cet exemple affiche tous les noms de fichiers du répertoire courant ayant pour extension `.txt` :


```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

`fnmatch.fnmatchcase(filename, pattern)`

Teste si *filename* correspond au motif *pattern*, en renvoyant *True* ou *False*; la comparaison est sensible à la casse et n'utilise pas la fonction `os.path.normcase()`.

`fnmatch.filter(names, pattern)`

Renvoie un sous-ensemble de la liste *names* correspondant au motif *pattern*. Similaire à `[n for n in names if fnmatch(n, pattern)]`, mais implémenté plus efficacement.

`fnmatch.translate(pattern)`

Renvoie le motif *pattern*, de style shell, converti en une expression régulière utilisable avec `re.match()`.

Exemple :

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\.\.txt)\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<_sre.SRE_Match object; span=(0, 10), match='foobar.txt'>
```

Voir aussi :

Module [*glob*](#) Recherche de chemins de style shell Unix

11.9 linecache — Accès direct aux lignes d'un texte

Code source : [Lib/linecache.py](#)

Le module *linecache* permet d'obtenir n'importe quelle ligne d'un fichier source Python. Le cas classique où de nombreuses lignes sont accédées est optimisé en utilisant un cache interne. C'est utilisé par le module *traceback* pour récupérer les lignes à afficher dans les piles d'appels.

Les fichiers sont ouverts par la fonction `tokenize.open()`. Cette fonction utilise `tokenize.detect_encoding()` pour obtenir l'encodage du fichier. En l'absence d'un BOM et d'un cookie d'encodage, c'est l'encodage UTF-8 qui sera utilisé.

Le module *linecache* définit les fonctions suivantes :

`linecache.getline(filename, lineno, module_globals=None)`

Récupère la ligne *lineno* du fichier *filename*. Cette fonction ne lèvera jamais d'exception, elle préférera renvoyer ' ' en cas d'erreur (le caractère de retour à la ligne sera inclus pour les lignes existantes).

Si le fichier *filename* n'est pas trouvé, la fonction le cherchera dans les chemins de recherche de modules, `sys.path`, après avoir vérifié si un `__loader__` (de la [PEP 302](#)) se trouve dans *module_globals*, dans le cas où le module a été importé depuis un fichier zip, ou une autre source hors du système de fichier.

`linecache.clearcache()`

Nettoie le cache. Utilisez cette fonction si vous n'avez plus besoin des lignes des fichiers précédemment lus via `getline()`.

`linecache.checkcache(filename=None)`

Vérifie la validité du cache. Utilisez cette fonction si les fichiers du cache pourraient avoir changé sur le disque, et que vous en voudriez une version à jour. Sans `filename`, toutes les entrées du cache seront vérifiées.

`linecache.lazycache(filename, module_globals)`

Récupère suffisamment d'informations sur un module situé hors du système de fichiers pour récupérer ses lignes plus tard via `getline()`, même si `module_globals` devient `None`. Cela évite de lire le fichier avant d'avoir besoin d'une ligne, tout en évitant de conserver les globales du module indéfiniment.

Nouveau dans la version 3.5.

Exemple :

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.10 `shutil` — Opérations de haut niveau sur les fichiers

Code source : [Lib/shutil.py](#)

Le module `shutil` propose des opérations de haut niveau sur les fichiers et ensembles de fichiers. En particulier, des fonctions pour copier et déplacer les fichiers sont proposées. Pour les opérations individuelles sur les fichiers, reportez-vous au module `os`.

Avertissement : Même les fonctions de copie haut niveau (`shutil.copy()`, `shutil.copy2()`) ne peuvent copier toutes les métadonnées des fichiers.

Sur les plateformes POSIX, cela signifie que le propriétaire et le groupe du fichier sont perdus, ainsi que les *ACLs*. Sur Mac OS, le clonage de ressource et autres métadonnées ne sont pas utilisés. Cela signifie que les ressources seront perdues et que le type de fichier et les codes créateur ne seront pas corrects. Sur Windows, les propriétaires des fichiers, *ACLs* et flux de données alternatifs ne sont pas copiés.

11.10.1 Opérations sur les répertoires et les fichiers

`shutil.copyfileobj(fsrc, fdst[, length])`

Copie le contenu de l'objet fichier `fsrc` dans l'objet fichier `fdst`. L'entier `length`, si spécifié, est la taille du tampon. En particulier, une valeur de `length` négative signifie la copie des données sans découper la source en morceaux ; par défaut les données sont lues par morceaux pour éviter la consommation mémoire non-contrôlée. À noter que si la position courante dans l'objet `fsrc` n'est pas 0, seul le contenu depuis la position courante jusqu'à la fin est copié.

`shutil.copyfile(src, dst, *, follow_symlinks=True)`

Copie le contenu (sans métadonnées) du fichier nommé `src` dans un fichier nommé `dst` et renvoie `dst`. `src` et `dst` sont des chemins sous forme de chaînes de caractères. `dst` doit être le chemin complet de la cible ; voir dans `shutil.copy()` pour une copie acceptant le chemin du dossier cible. Si `src` et `dst` désignent le même fichier `SameFileError` est levée.

La cible doit être accessible en écriture, sinon l'exception `OSError` est levée. Si `dst` existe déjà, il est remplacé. Les fichiers spéciaux comme les périphériques caractères ou bloc ainsi que les tubes (*pipes*) ne peuvent pas être copiés avec cette fonction.

Si `follow_symlinks` est faux et `src` est un lien symbolique, un nouveau lien symbolique est créé au lieu de copier le fichier pointé par `src`.

Modifié dans la version 3.3 : `IOError` était levée au lieu de `OSError`. Ajout de l'argument `follow_symlinks`. Maintenant renvoie `dst`.

Modifié dans la version 3.4 : Lève `SameFileError` au lieu de `Error`. Puisque la première est une sous-classe de la seconde, le changement assure la rétrocompatibilité.

exception `shutil.SameFileError`

Cette exception est levée si la source et la destination dans `copyfile()` sont le même fichier.

Nouveau dans la version 3.4.

`shutil.copymode(src, dst, *, follow_symlinks=True)`

Copie les octets de permission de `src` vers `dst`. Le contenu du fichier, le propriétaire et le groupe ne sont pas modifiés. `src` et `dst` sont des chaînes spécifiant les chemins. Si `follow_symlinks` est faux, et `src` et `dst` sont des liens symboliques, `copymode()` tente de modifier le mode de `dst` (au lieu du fichier vers lequel il pointe). Cette fonctionnalité n'est pas disponible sur toutes les plateformes ; voir `copystat()` pour plus d'informations. Si `copymode()` ne peut pas modifier les liens symboliques sur la plateforme cible alors que c'est demandé, il ne fait rien.

Modifié dans la version 3.3 : L'argument `follow_symlinks` a été ajouté.

`shutil.copystat(src, dst, *, follow_symlinks=True)`

Copie les bits définissant les droits d'accès, la date du dernier accès, de la dernière modification et les drapeaux (*flags* en anglais) de `src` vers `dst`. Sur Linux, `copystat()` copie également, si possible, les « *extended attributes* ». Le contenu du fichier, le propriétaire et le groupe ne sont pas affectés. `src` et `dst` sont des chaînes spécifiant les chemins.

Si `follow_symlinks` est faux et `src` et `dst` représentent des liens symboliques, `copystat()` agit sur les liens symboliques au lieu des fichiers cibles — elle lit les informations du lien symbolique de `src` et les écrit vers la destination pointée par `dst`.

Note : Toutes les plateformes n'offrent pas la possibilité d'examiner et modifier les liens symboliques. Python peut vous informer des fonctionnalités effectivement disponibles.

- Si `os.chmod` in `os.supports_follow_symlinks` est True, `copystat()` peut modifier les octets de droits d'accès du lien symbolique.
- If `os.utime` in `os.supports_follow_symlinks` is True, `copystat()` can modify the last access and modification times of a symbolic link.
- If `os.chflags` in `os.supports_follow_symlinks` is True, `copystat()` can modify the flags of a symbolic link. (`os.chflags` is not available on all platforms.)

On platforms where some or all of this functionality is unavailable, when asked to modify a symbolic link, `copystat()` will copy everything it can. `copystat()` never returns failure.

Please see `os.supports_follow_symlinks` for more information.

Modifié dans la version 3.3 : Added `follow_symlinks` argument and support for Linux extended attributes.

`shutil.copy(src, dst, *, follow_symlinks=True)`

Copies the file `src` to the file or directory `dst`. `src` and `dst` should be strings. If `dst` specifies a directory, the file will be copied into `dst` using the base filename from `src`. Returns the path to the newly created file.

If `follow_symlinks` is false, and `src` is a symbolic link, `dst` will be created as a symbolic link. If `follow_symlinks` is true and `src` is a symbolic link, `dst` will be a copy of the file `src` refers to.

`copy()` copies the file data and the file's permission mode (see `os.chmod()`). Other metadata, like the file's creation and modification times, is not preserved. To preserve all file metadata from the original, use `copy2()` instead.

Modifié dans la version 3.3 : Added `follow_symlinks` argument. Now returns path to the newly created file.

`shutil.copy2(src, dst, *, follow_symlinks=True)`

Identical to `copy()` except that `copy2()` also attempts to preserve file metadata.

When `follow_symlinks` is false, and `src` is a symbolic link, `copy2()` attempts to copy all metadata from the `src` symbolic link to the newly-created `dst` symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, `copy2()` will preserve all the metadata it can; `copy2()` never returns failure.

`copy2()` uses `copystat()` to copy the file metadata. Please see `copystat()` for more information about platform support for modifying symbolic link metadata.

Modifié dans la version 3.3 : Added `follow_symlinks` argument, try to copy extended file system attributes too (currently Linux only). Now returns path to the newly created file.

`shutil.ignore_patterns(*patterns)`

This factory function creates a function that can be used as a callable for `copytree()`'s `ignore` argument, ignoring files and directories that match one of the glob-style `patterns` provided. See the example below.

`shutil.copytree(src, dst, symlinks=False, ignore=None, copy_function=copy2, ignore_dangling_symlinks=False)`

Recursively copy an entire directory tree rooted at `src`, returning the destination directory. The destination directory, named by `dst`, must not already exist; it will be created as well as missing parent directories. Permissions and times of directories are copied with `copystat()`, individual files are copied using `shutil.copy2()`.

If `symlinks` is true, symbolic links in the source tree are represented as symbolic links in the new tree and the metadata of the original links will be copied as far as the platform allows; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

When `symlinks` is false, if the file pointed by the symlink doesn't exist, an exception will be added in the list of errors raised in an `Error` exception at the end of the copy process. You can set the optional `ignore_dangling_symlinks` flag to true if you want to silence this exception. Notice that this option has no effect on platforms that don't support `os.symlink()`.

If `ignore` is given, it must be a callable that will receive as its arguments the directory being visited by `copytree()`, and a list of its contents, as returned by `os.listdir()`. Since `copytree()` is called recursively, the `ignore` callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument); these names will then be ignored in the copy process. `ignore_patterns()` can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an `Error` is raised with a list of reasons.

If `copy_function` is given, it must be a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, `shutil.copy2()` is used, but any function that supports the same signature (like `shutil.copy()`) can be used.

Modifié dans la version 3.3 : Copy metadata when `symlinks` is false. Now returns `dst`.

Modifié dans la version 3.2 : Added the `copy_function` argument to be able to provide a custom copy function. Added the `ignore_dangling_symlinks` argument to silent dangling symlinks errors when `symlinks` is false.

`shutil.rmtree(path, ignore_errors=False, onerror=None)`

Delete an entire directory tree; `path` must point to a directory (but not a symbolic link to a directory). If `ignore_errors` is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by `onerror` or, if that is omitted, they raise an exception.

Note : On platforms that support the necessary fd-based functions a symlink attack resistant version of `rmtree()` is used by default. On other platforms, the `rmtree()` implementation is susceptible to a symlink attack : given proper timing and circumstances, attackers can manipulate symlinks on the filesystem to delete files they wouldn't be able to access otherwise. Applications can use the `rmtree.avoids_symlink_attacks` function attribute to determine which case applies.

If `onerror` is provided, it must be a callable that accepts three parameters : `function`, `path`, and `excinfo`.

The first parameter, *function*, is the function which raised the exception ; it depends on the platform and implementation. The second parameter, *path*, will be the path name passed to *function*. The third parameter, *excinfo*, will be the exception information returned by `sys.exc_info()`. Exceptions raised by *onerror* will not be caught.

Modifié dans la version 3.3 : Added a symlink attack resistant version that is used automatically if platform supports fd-based functions.

`rmtree.avoids_symlink_attacks`

Indicates whether the current platform and implementation provides a symlink attack resistant version of `rmtree()`. Currently this is only true for platforms supporting fd-based directory access functions.

Nouveau dans la version 3.3.

`shutil.move(src, dst, copy_function=copy2)`

Recursively move a file or directory (*src*) to another location (*dst*) and return the destination.

If the destination is an existing directory, then *src* is moved inside that directory. If the destination already exists but is not a directory, it may be overwritten depending on `os.rename()` semantics.

If the destination is on the current filesystem, then `os.rename()` is used. Otherwise, *src* is copied to *dst* using *copy_function* and then removed. In case of symlinks, a new symlink pointing to the target of *src* will be created in or as *dst* and *src* will be removed.

If *copy_function* is given, it must be a callable that takes two arguments *src* and *dst*, and will be used to copy *src* to *dst* if `os.rename()` cannot be used. If the source is a directory, `copytree()` is called, passing it the *copy_function()*. The default *copy_function* is `copy2()`. Using `copy()` as the *copy_function* allows the move to succeed when it is not possible to also copy the metadata, at the expense of not copying any of the metadata.

Modifié dans la version 3.3 : Added explicit symlink handling for foreign filesystems, thus adapting it to the behavior of GNU's `mv`. Now returns *dst*.

Modifié dans la version 3.5 : Added the *copy_function* keyword argument.

`shutil.disk_usage(path)`

Return disk usage statistics about the given path as a *named tuple* with the attributes *total*, *used* and *free*, which are the amount of total, used and free space, in bytes. On Windows, *path* must be a directory ; on Unix, it can be a file or directory.

Nouveau dans la version 3.3.

Disponibilité : Unix, Windows.

`shutil.chown(path, user=None, group=None)`

Change owner *user* and/or *group* of the given *path*.

user can be a system user name or a uid ; the same applies to *group*. At least one argument is required.

See also `os.chown()`, the underlying function.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)`

Return the path to an executable which would be run if the given *cmd* was called. If no *cmd* would be called, return *None*.

mode is a permission mask passed to `os.access()`, by default determining if the file exists and executable.

When no *path* is specified, the results of `os.environ()` are used, returning either the « PATH » value or a fallback of `os.defpath`.

On Windows, the current directory is always prepended to the *path* whether or not you use the default or provide your own, which is the behavior the command shell uses when finding executables. Additionally, when finding the *cmd* in the *path*, the `PATHEXT` environment variable is checked. For example, if you call `shutil.which("python")`, `which()` will search `PATHEXT` to know that it should look for `python.exe` within the *path* directories. For example, on Windows :

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

Nouveau dans la version 3.3.

exception `shutil.Error`

This exception collects exceptions that are raised during a multi-file operation. For `copytree()`, the exception argument is a list of 3-tuples (*srcname*, *dstname*, *exception*).

copytree example

This example is the implementation of the `copytree()` function, described above, with the docstring omitted. It demonstrates many of the other functions provided by this module.

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
                # XXX What about devices, sockets etc.?
        except OSError as why:
            errors.append((srcname, dstname, str(why)))
        # catch the Error from the recursive copytree so that we can
        # continue with other files
        except Error as err:
            errors.extend(err.args[0])
    try:
        copystat(src, dst)
    except OSError as why:
        # can't copy file access times on Windows
        if why.winerror is None:
            errors.extend((src, dst, str(why)))
    if errors:
        raise Error(errors)
```

Another example that uses the `ignore_patterns()` helper :

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

This will copy everything except `.pyc` files and files or directories whose name starts with `tmp`.

Another example that uses the `ignore` argument to add a logging call :

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return [] # nothing will be ignored
```

(suite sur la page suivante)

(suite de la page précédente)

```
copytree(source, destination, ignore=_logpath)
```

rmtree example

This example shows how to remove a directory tree on Windows where some of the files have their read-only bit set. It uses the `onerror` callback to clear the readonly bit and reattempt the remove. Any subsequent failure will propagate.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```

11.10.2 Archiving operations

Nouveau dans la version 3.2.

Modifié dans la version 3.5 : Added support for the *xz*tar format.

High-level utilities to create and read compressed and archived files are also provided. They rely on the *zipfile* and *tarfile* modules.

```
shutil.make_archive(base_name, format[, root_dir[, base_dir[, verbose[, dry_run[, owner[, group[,
                        logger]]]]]])
```

Create an archive file (such as zip or tar) and return its name.

base_name is the name of the file to create, including the path, minus any format-specific extension. *format* is the archive format : one of « zip » (if the *zlib* module is available), « tar », « gztar » (if the *zlib* module is available), « bztar » (if the *bz2* module is available), or « xztar » (if the *lzma* module is available).

root_dir is a directory that will be the root directory of the archive ; for example, we typically `chdir` into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from ; i.e. *base_dir* will be the common prefix of all files and directories in the archive.

root_dir and *base_dir* both default to the current directory.

If *dry_run* is true, no archive is created, but the operations that would be executed are logged to *logger*.

owner and *group* are used when creating a tar archive. By default, uses the current owner and group.

logger must be an object compatible with **PEP 282**, usually an instance of *logging.Logger*.

The *verbose* argument is unused and deprecated.

```
shutil.get_archive_formats()
```

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple (name, description).

By default *shutil* provides these formats :

- *zip* : ZIP file (if the *zlib* module is available).
- *tar* : uncompressed tar file.
- *gztar* : gzip'ed tar-file (if the *zlib* module is available).
- *bztar* : bzip2'ed tar-file (if the *bz2* module is available).
- *xztar* : xz'ed tar-file (if the *lzma* module is available).

You can register new formats or provide your own archiver for any existing formats, by using `register_archive_format()`.

`shutil.register_archive_format(name, function[, extra_args[, description]])`

Register an archiver for the format *name*.

function is the callable that will be used to unpack archives. The callable will receive the *base_name* of the file to create, followed by the *base_dir* (which defaults to `os.curdir`) to start archiving from. Further arguments are passed as keyword arguments : *owner*, *group*, *dry_run* and *logger* (as passed in `make_archive()`).

If given, *extra_args* is a sequence of (*name*, *value*) pairs that will be used as extra keywords arguments when the archiver callable is used.

description is used by `get_archive_formats()` which returns the list of archivers. Defaults to an empty string.

`shutil.unregister_archive_format(name)`

Remove the archive format *name* from the list of supported formats.

`shutil.unpack_archive(filename[, extract_dir[, format]])`

Unpack an archive. *filename* is the full path of the archive.

extract_dir is the name of the target directory where the archive is unpacked. If not provided, the current working directory is used.

format is the archive format : one of « zip », « tar », « gztar », « bztar », or « xztar ». Or any other format registered with `register_unpack_format()`. If not provided, `unpack_archive()` will use the archive file name extension and see if an unpacker was registered for that extension. In case none is found, a `ValueError` is raised.

`shutil.register_unpack_format(name, extensions, function[, extra_args[, description]])`

Registers an unpack format. *name* is the name of the format and *extensions* is a list of extensions corresponding to the format, like `.zip` for Zip files.

function is the callable that will be used to unpack archives. The callable will receive the path of the archive, followed by the directory the archive must be extracted to.

When provided, *extra_args* is a sequence of (*name*, *value*) tuples that will be passed as keywords arguments to the callable.

description can be provided to describe the format, and will be returned by the `get_unpack_formats()` function.

`shutil.unregister_unpack_format(name)`

Unregister an unpack format. *name* is the name of the format.

`shutil.get_unpack_formats()`

Return a list of all registered formats for unpacking. Each element of the returned sequence is a tuple (*name*, *extensions*, *description*).

By default `shutil` provides these formats :

- *zip* : ZIP file (unpacking compressed files works only if the corresponding module is available).
- *tar* : uncompressed tar file.
- *gztar* : gzip'ed tar-file (if the `zlib` module is available).
- *bztar* : bzip2'ed tar-file (if the `bz2` module is available).
- *xztar* : xz'ed tar-file (if the `lzma` module is available).

You can register new formats or provide your own unpacker for any existing formats, by using `register_unpack_format()`.

Archiving example

In this example, we create a gzip'ed tar-file archive containing all files found in the `.ssh` directory of the user :

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

The resulting archive contains :

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx----- tarek/staff      0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff    609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff     65 2008-06-09 13:26:54 ./config
-rwx----- tarek/staff    668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff    609 2008-06-09 13:26:54 ./id_dsa.pub
-rw----- tarek/staff   1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff    397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff  37192 2010-02-06 18:23:10 ./known_hosts
```

11.10.3 Querying the size of the output terminal

`shutil.get_terminal_size (fallback=(columns, lines))`

Get the size of the terminal window.

For each of the two dimensions, the environment variable, `COLUMNS` and `LINES` respectively, is checked. If the variable is defined and the value is a positive integer, it is used.

When `COLUMNS` or `LINES` is not defined, which is the common case, the terminal connected to `sys.__stdout__` is queried by invoking `os.get_terminal_size()`.

If the terminal size cannot be successfully queried, either because the system doesn't support querying, or because we are not connected to a terminal, the value given in `fallback` parameter is used. `fallback` defaults to `(80, 24)` which is the default size used by many terminal emulators.

The value returned is a named tuple of type `os.terminal_size`.

See also : The Single UNIX Specification, Version 2, [Other Environment Variables](#).

Nouveau dans la version 3.3.

11.11 macpath — Fonctions de manipulation de chemins pour Mac OS 9

Code source : [Lib/macpath.py](#)

Ce module est une implémentation du module `os.path` pour Mac OS 9 (ou plus ancien). Il peut être utilisé pour manipuler des chemins dans l'ancien style Macintosh sur Mac OS X (ou n'importe quelle autre plateforme).

Les fonctions suivantes sont disponibles dans ce module : `normcase()`, `normpath()`, `isabs()`, `join()`, `split()`, `isdir()`, `isfile()`, `walk()`, `exists()`. Toutes les autres fonctions d'`os.path` sont aussi disponibles, mais vides, pour garder la compatibilité.

Voir aussi :

Module *os* Interfaces du système d'exploitation, incluant des fonctions pour travailler avec des fichiers dans un niveau plus bas que les *objets fichiers* de Python.

Module *io* Bibliothèque d'entrées/sorties native de Python, incluant des classes abstraites et concrètes tel que les I/O sur les fichiers.

Fonction native *open ()* Le moyen classique pour ouvrir des fichiers pour les lire ou y écrire avec Python.

Persistence des données

Les modules décrits dans ce chapitre permettent de stocker des données Python de manière persistante typiquement sur disque. Les modules *pickle* et *marshal* peuvent transformer n'importe quel type Python en une séquence d'octets, puis recréer les objets depuis ces octets. Les différents modules du paquet *dbm* gèrent une catégorie de formats de fichier basée sur des hash, stockant des correspondances entre chaînes de caractères.

La liste des modules documentés dans ce chapitre est :

12.1 *pickle* — Module de sérialisation d'objets Python

Code source : [Lib/pickle.py](#)

Le module *pickle* implémente des protocoles binaires de sérialisation et dé-sérialisation d'objets Python. Le *pickling* est le procédé par lequel une hiérarchie d'objets Python est convertie en flux d'octets. *unpickling* est l'opération inverse, par laquelle un flux d'octets (à partir d'un *binary file* ou *bytes-like object*) est converti en hiérarchie d'objets. *Pickling* (et *unpickling*) sont alternativement connus sous les termes de « sérialisation », de « *marshalling* »¹ ou encore de « *flattening* ». Cependant pour éviter la confusion les termes utilisés ici sont *pickling* et *unpickling*.

Avertissement : Le module *pickle* n'est pas sécurisé contre les données erronées et malicieusement construites. Ne jamais *unpickle* la donnée reçue à partir d'une source non fiable ou non authentifiée.

1. Don't confuse this with the *marshal* module

12.1.1 Relations aux autres modules python

Comparaison avec `marshal`

Python possède un module de bas niveau en sérialisation appelé `marshal`, mais en général il est préférable d'utiliser `pickle` pour sérialiser des objets Python. `marshal` existe principalement pour gérer les fichiers Python en `.pyc`.

Le module `pickle` diffère du module `marshal` sur plusieurs aspects :

- Le module `pickle` garde la trace des objets qu'il a déjà sérialisés, pour faire en sorte que les prochaines références à cet objet ne soient pas sérialisées à nouveau. `marshal` ne le fait pas.
Ça a des implications sur les objets partagés et les objets récurifs. Les objets récurifs sont des objets qui contiennent des références à eux-mêmes. Ceux-ci ne sont pas gérées par `marshal` : lui donner un objet récurif va le faire planter. Un objet est partagé lorsque que plusieurs références pointent dessus, depuis différents endroits dans la hiérarchie sérialisée. Le module `pickle` repère ces partages et ne stocke ces objets qu'une seule fois. Les objets partagés restent ainsi partagés, ce qui peut être très important pour les objets muables.
- `marshal` ne peut être utilisé pour la sérialisation et l'instanciation de classes définies par les utilisateurs. `pickle` peut sauvegarder et restaurer les instances de classes de manière transparente. Cependant la définition de classe doit être importable et lancée dans le même module et de la même manière que lors de son importation.
- The `marshal` serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support `.pyc` files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The `pickle` serialization format is guaranteed to be backwards compatible across Python releases.

Comparison with `json`

There are fundamental differences between the pickle protocols and **JSON (JavaScript Object Notation)** :

- JSON is a text serialization format (it outputs unicode text, although most of the time it is then encoded to `utf-8`), while pickle is a binary serialization format ;
- JSON is human-readable, while pickle is not ;
- JSON is interoperable and widely used outside of the Python ecosystem, while pickle is Python-specific ;
- JSON, by default, can only represent a subset of the Python built-in types, and no custom classes ; pickle can represent an extremely large number of Python types (many of them automatically, by clever usage of Python's introspection facilities ; complex cases can be tackled by implementing *specific object APIs*).

Voir aussi :

The `json` module : a standard library module allowing JSON serialization and deserialization.

12.1.2 Data stream format

The data format used by `pickle` is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as JSON or XDR (which can't represent pointer sharing) ; however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the `pickle` data format uses a relatively compact binary representation. If you need optimal size characteristics, you can efficiently *compress* pickled data.

The module `pickletools` contains tools for analyzing data streams generated by `pickle`. `pickletools` source code has extensive comments about opcodes used by pickle protocols.

There are currently 5 different protocols which can be used for pickling. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

- Protocol version 0 is the original « human-readable » protocol and is backwards compatible with earlier versions of Python.
- Protocol version 1 is an old binary format which is also compatible with earlier versions of Python.

- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of *new-style classes*. Refer to [PEP 307](#) for information about improvements brought by protocol 2.
- Protocol version 3 was added in Python 3.0. It has explicit support for *bytes* objects and cannot be unpickled by Python 2.x. This is the default protocol, and the recommended protocol when compatibility with other Python 3 versions is required.
- Protocol version 4 was added in Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data format optimizations. Refer to [PEP 3154](#) for information about improvements brought by protocol 4.

Note : Serialization is a more primitive notion than persistence ; although *pickle* reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) issue of concurrent access to persistent objects. The *pickle* module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The *shelve* module provides a simple interface to pickle and unpickle objects on DBM-style database files.

12.1.3 Module Interface

To serialize an object hierarchy, you simply call the *dumps()* function. Similarly, to de-serialize a data stream, you call the *loads()* function. However, if you want more control over serialization and de-serialization, you can create a *Pickler* or an *Unpickler* object, respectively.

The *pickle* module provides the following constants :

`pickle.HIGHEST_PROTOCOL`

An integer, the highest *protocol version* available. This value can be passed as a *protocol* value to functions *dump()* and *dumps()* as well as the *Pickler* constructor.

`pickle.DEFAULT_PROTOCOL`

An integer, the default *protocol version* used for pickling. May be less than *HIGHEST_PROTOCOL*. Currently the default protocol is 3, a new protocol designed for Python 3.

The *pickle* module provides the following functions to make the pickling process more convenient :

`pickle.dump(obj, file, protocol=None, *, fix_imports=True)`

Write a pickled representation of *obj* to the open *file object file*. This is equivalent to `Pickler(file, protocol).dump(obj)`.

The optional *protocol* argument, an integer, tells the pickler to use the given protocol ; supported protocols are 0 to *HIGHEST_PROTOCOL*. If not specified, the default is *DEFAULT_PROTOCOL*. If a negative number is specified, *HIGHEST_PROTOCOL* is selected.

The *file* argument must have a `write()` method that accepts a single bytes argument. It can thus be an on-disk file opened for binary writing, an *io.BytesIO* instance, or any other custom object that meets this interface.

If *fix_imports* is true and *protocol* is less than 3, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

`pickle.dumps(obj, protocol=None, *, fix_imports=True)`

Return the pickled representation of the object as a *bytes* object, instead of writing it to a file.

Arguments *protocol* and *fix_imports* have the same meaning as in *dump()*.

`pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict")`

Read a pickled object representation from the open *file object file* and return the reconstituted object hierarchy specified therein. This is equivalent to `Unpickler(file).load()`.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object's representation are ignored.

The argument *file* must have two methods, a `read()` method that takes an integer argument, and a `readline()` method that requires no arguments. Both methods should return bytes. Thus *file* can be an on-disk file opened for binary reading, an `io.BytesIO` object, or any other custom object that meets this interface.

Optional keyword arguments are *fix_imports*, *encoding* and *errors*, which are used to control compatibility support for pickle stream generated by Python 2. If *fix_imports* is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The *encoding* and *errors* tell pickle how to decode 8-bit string instances pickled by Python 2; these default to “ASCII” and “strict”, respectively. The *encoding* can be “bytes” to read these 8-bit string instances as bytes objects. Using `encoding='latin1'` is required for unpickling NumPy arrays and instances of *datetime*, *date* and *time* pickled by Python 2.

`pickle.loads(bytes_object, *, fix_imports=True, encoding="ASCII", errors="strict")`

Read a pickled object hierarchy from a *bytes* object and return the reconstituted object hierarchy specified therein. The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object’s representation are ignored.

Optional keyword arguments are *fix_imports*, *encoding* and *errors*, which are used to control compatibility support for pickle stream generated by Python 2. If *fix_imports* is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The *encoding* and *errors* tell pickle how to decode 8-bit string instances pickled by Python 2; these default to “ASCII” and “strict”, respectively. The *encoding* can be “bytes” to read these 8-bit string instances as bytes objects. Using `encoding='latin1'` is required for unpickling NumPy arrays and instances of *datetime*, *date* and *time* pickled by Python 2.

The *pickle* module defines three exceptions :

exception `pickle.PickleError`

Common base class for the other pickling exceptions. It inherits *Exception*.

exception `pickle.PicklingError`

Error raised when an unpicklable object is encountered by *Pickler*. It inherits *PickleError*.

Refer to *What can be pickled and unpickled?* to learn what kinds of objects can be pickled.

exception `pickle.UnpicklingError`

Error raised when there is a problem unpickling an object, such as a data corruption or a security violation. It inherits *PickleError*.

Note that other exceptions may also be raised during unpickling, including (but not necessarily limited to) *AttributeError*, *EOFError*, *ImportError*, and *IndexError*.

The *pickle* module exports two classes, *Pickler* and *Unpickler* :

class `pickle.Pickler(file, protocol=None, *, fix_imports=True)`

This takes a binary file for writing a pickle data stream.

The optional *protocol* argument, an integer, tells the pickler to use the given protocol; supported protocols are 0 to *HIGHEST_PROTOCOL*. If not specified, the default is *DEFAULT_PROTOCOL*. If a negative number is specified, *HIGHEST_PROTOCOL* is selected.

The *file* argument must have a `write()` method that accepts a single bytes argument. It can thus be an on-disk file opened for binary writing, an `io.BytesIO` instance, or any other custom object that meets this interface.

If *fix_imports* is true and *protocol* is less than 3, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

dump (*obj*)

Write a pickled representation of *obj* to the open file object given in the constructor.

persistent_id (*obj*)

Do nothing by default. This exists so a subclass can override it.

If *persistent_id()* returns *None*, *obj* is pickled as usual. Any other value causes *Pickler* to emit the returned value as a persistent ID for *obj*. The meaning of this persistent ID should be defined by *Unpickler.persistent_load()*. Note that the value returned by *persistent_id()* cannot itself have a persistent ID.

See *Persistence of External Objects* for details and examples of uses.

dispatch_table

A pickler object's dispatch table is a registry of *reduction functions* of the kind which can be declared using `copyreg.pickle()`. It is a mapping whose keys are classes and whose values are reduction functions. A reduction function takes a single argument of the associated class and should conform to the same interface as a `__reduce__()` method.

By default, a pickler object will not have a `dispatch_table` attribute, and it will instead use the global dispatch table managed by the `copyreg` module. However, to customize the pickling for a specific pickler object one can set the `dispatch_table` attribute to a dict-like object. Alternatively, if a subclass of `Pickler` has a `dispatch_table` attribute then this will be used as the default dispatch table for instances of that class.

See *Dispatch Tables* for usage examples.

Nouveau dans la version 3.3.

fast

Deprecated. Enable fast mode if set to a true value. The fast mode disables the usage of memo, therefore speeding the pickling process by not generating superfluous PUT opcodes. It should not be used with self-referential objects, doing otherwise will cause `Pickler` to recurse infinitely.

Use `pickletools.optimize()` if you need more compact pickles.

class `pickle.Unpickler` (*file*, *, *fix_imports*=True, *encoding*="ASCII", *errors*="strict")

This takes a binary file for reading a pickle data stream.

The protocol version of the pickle is detected automatically, so no protocol argument is needed.

The argument *file* must have two methods, a `read()` method that takes an integer argument, and a `readline()` method that requires no arguments. Both methods should return bytes. Thus *file* can be an on-disk file object opened for binary reading, an `io.BytesIO` object, or any other custom object that meets this interface.

Optional keyword arguments are *fix_imports*, *encoding* and *errors*, which are used to control compatibility support for pickle stream generated by Python 2. If *fix_imports* is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The *encoding* and *errors* tell pickle how to decode 8-bit string instances pickled by Python 2; these default to "ASCII" and "strict", respectively. The *encoding* can be "bytes" to read these 8-bit string instances as bytes objects.

load()

Read a pickled object representation from the open file object given in the constructor, and return the re-constituted object hierarchy specified therein. Bytes past the pickled object's representation are ignored.

persistent_load (*pid*)

Raise an `UnpicklingError` by default.

If defined, `persistent_load()` should return the object specified by the persistent ID *pid*. If an invalid persistent ID is encountered, an `UnpicklingError` should be raised.

See *Persistence of External Objects* for details and examples of uses.

find_class (*module*, *name*)

Import *module* if necessary and return the object called *name* from it, where the *module* and *name* arguments are `str` objects. Note, unlike its name suggests, `find_class()` is also used for finding functions.

Subclasses may override this to gain control over what type of objects and how they can be loaded, potentially reducing security risks. Refer to *Restricting Globals* for details.

12.1.4 What can be pickled and unpickled ?

The following types can be pickled :

- None, True, and False
- integers, floating point numbers, complex numbers
- strings, bytes, bytearray
- tuples, lists, sets, and dictionaries containing only picklable objects
- functions defined at the top level of a module (using `def`, not `lambda`)
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module
- instances of such classes whose `__dict__` or the result of calling `__getstate__()` is picklable (see section *Pickling Class Instances* for details).

Attempts to pickle unpicklable objects will raise the `PicklingError` exception; when this happens, an unspecified number of bytes may have already been written to the underlying file. Trying to pickle a highly recursive data structure may exceed the maximum recursion depth, a `RecursionError` will be raised in this case. You can carefully raise this limit with `sys.setrecursionlimit()`.

Note that functions (built-in and user-defined) are pickled by « fully qualified » name reference, not by value.² This means that only the function name is pickled, along with the name of the module the function is defined in. Neither the function's code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised.³

Similarly, classes are pickled by named reference, so the same restrictions in the unpickling environment apply. Note that none of the class's code or data is pickled, so in the following example the class attribute `attr` is not restored in the unpickling environment :

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

These restrictions are why picklable functions and classes must be defined in the top level of a module.

Similarly, when class instances are pickled, their class's code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods to the class and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's `__setstate__()` method.

12.1.5 Pickling Class Instances

In this section, we describe the general mechanisms available to you to define, customize, and control how class instances are pickled and unpickled.

In most cases, no additional code is needed to make instances picklable. By default, pickle will retrieve the class and the attributes of an instance via introspection. When a class instance is unpickled, its `__init__()` method is usually *not* invoked. The default behaviour first creates an uninitialized instance and then restores the saved attributes. The following code shows an implementation of this behaviour :

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def load(cls, attributes):
```

(suite sur la page suivante)

2. This is why `lambda` functions cannot be pickled : all `lambda` functions share the same name : `<lambda>`.

3. The exception raised will likely be an `ImportError` or an `AttributeError` but it could be something else.

(suite de la page précédente)

```
obj = cls.__new__(cls)
obj.__dict__.update(attributes)
return obj
```

Classes can alter the default behaviour by providing one or several special methods :

`object.__getnewargs_ex__()`

In protocols 2 and newer, classes that implements the `__getnewargs_ex__()` method can dictate the values passed to the `__new__()` method upon unpickling. The method must return a pair `(args, kwargs)` where `args` is a tuple of positional arguments and `kwargs` a dictionary of named arguments for constructing the object. Those will be passed to the `__new__()` method upon unpickling.

You should implement this method if the `__new__()` method of your class requires keyword-only arguments. Otherwise, it is recommended for compatibility to implement `__getnewargs__()`.

Modifié dans la version 3.6 : `__getnewargs_ex__()` is now used in protocols 2 and 3.

`object.__getnewargs__()`

This method serves a similar purpose as `__getnewargs_ex__()`, but supports only positional arguments. It must return a tuple of arguments `args` which will be passed to the `__new__()` method upon unpickling.

`__getnewargs__()` will not be called if `__getnewargs_ex__()` is defined.

Modifié dans la version 3.6 : Before Python 3.6, `__getnewargs__()` was called instead of `__getnewargs_ex__()` in protocols 2 and 3.

`object.__getstate__()`

Classes can further influence how their instances are pickled ; if the class defines the method `__getstate__()`, it is called and the returned object is pickled as the contents for the instance, instead of the contents of the instance's dictionary. If the `__getstate__()` method is absent, the instance's `__dict__` is pickled as usual.

`object.__setstate__(state)`

Upon unpickling, if the class defines `__setstate__()`, it is called with the unpickled state. In that case, there is no requirement for the state object to be a dictionary. Otherwise, the pickled state must be a dictionary and its items are assigned to the new instance's dictionary.

Note : If `__getstate__()` returns a false value, the `__setstate__()` method will not be called upon unpickling.

Refer to the section *Handling Stateful Objects* for more information about how to use the methods `__getstate__()` and `__setstate__()`.

Note : At unpickling time, some methods like `__getattr__()`, `__getattribute__()`, or `__setattr__()` may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement `__getnewargs__()` or `__getnewargs_ex__()` to establish such an invariant; otherwise, neither `__new__()` nor `__init__()` will be called.

As we shall see, pickle does not use directly the methods described above. In fact, these methods are part of the copy protocol which implements the `__reduce__()` special method. The copy protocol provides a unified interface for retrieving the data necessary for pickling and copying objects.⁴

Although powerful, implementing `__reduce__()` directly in your classes is error prone. For this reason, class designers should use the high-level interface (i.e., `__getnewargs_ex__()`, `__getstate__()` and `__setstate__()`) whenever possible. We will show, however, cases where using `__reduce__()` is the only option or leads to more efficient pickling or both.

4. The `copy` module uses this protocol for shallow and deep copying operations.

`object.__reduce__()`

The interface is currently defined as follows. The `__reduce__()` method takes no argument and shall return either a string or preferably a tuple (the returned object is often referred to as the « reduce value »).

If a string is returned, the string should be interpreted as the name of a global variable. It should be the object's local name relative to its module; the pickle module searches the module namespace to determine the object's module. This behaviour is typically useful for singletons.

When a tuple is returned, it must be between two and five items long. Optional items can either be omitted, or `None` can be provided as their value. The semantics of each item are in order :

- A callable object that will be called to create the initial version of the object.
- A tuple of arguments for the callable object. An empty tuple must be given if the callable does not accept any argument.
- Optionally, the object's state, which will be passed to the object's `__setstate__()` method as previously described. If the object has no such method then, the value must be a dictionary and it will be added to the object's `__dict__` attribute.
- Optionally, an iterator (and not a sequence) yielding successive items. These items will be appended to the object either using `obj.append(item)` or, in batch, using `obj.extend(list_of_items)`. This is primarily used for list subclasses, but may be used by other classes as long as they have `append()` and `extend()` methods with the appropriate signature. (Whether `append()` or `extend()` is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)
- Optionally, an iterator (not a sequence) yielding successive key-value pairs. These items will be stored to the object using `obj[key] = value`. This is primarily used for dictionary subclasses, but may be used by other classes as long as they implement `__setitem__()`.

`object.__reduce_ex__(protocol)`

Alternatively, a `__reduce_ex__()` method may be defined. The only difference is this method should take a single integer argument, the protocol version. When defined, pickle will prefer it over the `__reduce__()` method. In addition, `__reduce__()` automatically becomes a synonym for the extended version. The main use for this method is to provide backwards-compatible reduce values for older Python releases.

Persistence of External Objects

For the benefit of object persistence, the `pickle` module supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a persistent ID, which should be either a string of alphanumeric characters (for protocol 0)⁵ or just an arbitrary object (for any newer protocol).

The resolution of such persistent IDs is not defined by the `pickle` module; it will delegate this resolution to the user defined methods on the pickler and unpickler, `persistent_id()` and `persistent_load()` respectively.

To pickle objects that have an external persistent id, the pickler must have a custom `persistent_id()` method that takes an object as an argument and returns either `None` or the persistent id for that object. When `None` is returned, the pickler simply pickles the object as normal. When a persistent ID string is returned, the pickler will pickle that object, along with a marker so that the unpickler will recognize it as a persistent ID.

To unpickle external objects, the unpickler must have a custom `persistent_load()` method that takes a persistent ID object and returns the referenced object.

Here is a comprehensive example presenting how persistent ID can be used to pickle external objects by reference.

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.

import pickle
import sqlite3
```

(suite sur la page suivante)

5. The limitation on alphanumeric characters is due to the fact the persistent IDs, in protocol 0, are delimited by the newline character. Therefore if any kind of newline characters occurs in persistent IDs, the resulting pickle will become unreadable.

(suite de la page précédente)

```

from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None

class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")

def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
    )
    for task in tasks:

```

(suite sur la page suivante)

(suite de la page précédente)

```

        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)

if __name__ == '__main__':
    main()

```

Dispatch Tables

If one wants to customize pickling of some classes without disturbing any other code which depends on pickling, then one can create a pickler with a private dispatch table.

The global dispatch table managed by the `copyreg` module is available as `copyreg.dispatch_table`. Therefore, one may choose to use a modified copy of `copyreg.dispatch_table` as a private dispatch table.

Par exemple

```

f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass

```

creates an instance of `pickle.Pickler` with a private dispatch table which handles the `SomeClass` class specially. Alternatively, the code

```

class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)

```

does the same, but all instances of `MyPickler` will by default share the same dispatch table. The equivalent code using the `copyreg` module is

```

copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)

```

Handling Stateful Objects

Here's an example that shows how to modify pickling behavior for a class. The `TextReader` class opens a text file, and returns the line number and line contents each time its `readline()` method is called. If a `TextReader` instance is pickled, all attributes *except* the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last location. The `__setstate__()` and `__getstate__()` methods are used to implement this behavior.

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file
```

A sample usage might be something like this :

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6 Restricting Globals

By default, unpickling will import any class or function that it finds in the pickle data. For many applications, this behaviour is unacceptable as it permits the unpickler to import and invoke arbitrary code. Just consider what this hand-crafted pickle data stream does when loaded :

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntr.")
hello world
0
```

In this example, the unpickler imports the `os.system()` function and then apply the string argument « echo hello world ». Although this example is inoffensive, it is not difficult to imagine one that could damage your system.

For this reason, you may want to control what gets unpickled by customizing `Unpickler.find_class()`. Unlike its name suggests, `Unpickler.find_class()` is called whenever a global (i.e., a class or a function) is requested. Thus it is possible to either completely forbid globals or restrict them to a safe subset.

Here is an example of an unpickler allowing only few safe classes from the `builtins` module to be loaded :

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                      (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

A sample usage of our unpickler working has intended :

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntr.")
Traceback (most recent call last):
...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'__import__\n("os"), "system")\'
...                  b'("echo hello world")\ntr.')
Traceback (most recent call last):
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

As our examples shows, you have to be careful with what you allow to be unpickled. Therefore if security is a concern, you may want to consider alternatives such as the marshalling API in *xmlrpc.client* or third-party solutions.

12.1.7 Performances

Recent versions of the pickle protocol (from protocol 2 and upwards) feature efficient binary encodings for several common features and built-in types. Also, the *pickle* module has a transparent optimizer written in C.

12.1.8 Exemples

For the simplest code, use the *dump()* and *load()* functions.

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

The following example reads the resulting pickled data.

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

Voir aussi :

Module *copyreg* Pickle interface constructor registration for extension types.

Module *pickletools* Tools for working with and analyzing pickled data.

Module *shelve* Indexed databases of objects; uses *pickle*.

Module *copy* Shallow and deep object copying.

Module *marshal* High-performance serialization of built-in types.

Notes

12.2 copyreg — Enregistre les fonctions support de pickle

Code source : [Lib/copyreg.py](#)

Le module `copyreg` permet de définir des fonctions utilisées durant la sérialisation avec `pickle` de certains objets. Les modules `pickle` et `copy` utilisent ces fonctions lors d'une sérialisation ou d'une copie de ces objets. Le module propose alors des informations de configuration à propos de constructeurs d'objets qui ne sont pas des classes. De tels constructeurs peuvent être des instances de classes ou des fonctions.

`copyreg.constructor(object)`

Déclare *object* comme étant un constructeur valide. Si *object* n'est pas callable (et n'est donc pas un constructeur valide), l'erreur `TypeError` est levée.

`copyreg.pickle(type, function, constructor=None)`

Déclare que *function* devrait être utilisée en tant que fonction de *réduction* pour des objets de type *type*. *function* doit soit retourner une chaîne de caractères soit un tuple qui contiens deux ou trois éléments.

Le paramètre optionnel *constructor*, s'il est donné, est un objet callable qui peut être utilisé pour reconstruire l'objet lorsqu'il est appelé avec un tuple d'arguments retournés par *function* durant la sérialisation avec `pickle`. Une exception `TypeError` sera levée si *object* est une classe ou si *constructor* n'est pas callable.

Voir le module `pickle` pour plus de détails sur l'interface attendue de *function* et *constructor*. Notez que l'attribut `dispatch_table` d'un objet pickler ou d'une sous-classe de `pickle.Pickler` peut aussi être utilisée pour déclarer des fonctions réductrices.

12.2.1 Exemple

L'exemple si-dessous essaye de démontrer comment enregistrer une fonction `pickle` et comment elle sera utilisée :

```
>>> import copyreg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```


12.3 `shelve` — Objet Python persistant

Code source : [Lib/shelve.py](#)

Un *shelf* est un objet persistant, dictionnaire-compatible. La différence avec les bases de données *dbm* est que les valeurs (pas les clés !) dans un *shelf* peuvent être des objets Python arbitraires — n'importe quoi que le module *pickle* peut gérer. Cela inclut la plupart des instances de classe, des types de données récursives, et les objets contenant beaucoup de sous-objets partagés. Les clés sont des chaînes de caractères ordinaires.

`shelve.open(filename, flag='c', protocol=None, writeback=False)`

Ouvre un dictionnaire persistant. Le nom de fichier spécifié est le nom de fichier sans (son) extension pour la base de données sous-jacente. Comme effet de bord, une extension peut être ajoutée au nom de fichier et plus d'un fichier peut être créé. Par défaut, le fichier de base de données sous-jacente est ouvert en lecture et en écriture. Le paramètre optionnel *flag* possède la même interprétation que le paramètre *flag* de `dbm.open()`.

Par défaut, les *pickles* de version 3 sont utilisés pour sérialiser des valeurs. La version du protocole de *pickle* peut être spécifiée avec le paramètre *protocol*.

À cause de la sémantique Python, un *shelf* ne peut pas savoir lorsqu'une entrée modifiable de dictionnaire persistant est modifiée. Par défaut les objets modifiés sont écrits *seulement* lorsqu'ils sont assignés à une *shelf* (voir [Exemple](#)). Si le paramètre optionnel *writeback* est mis à `True`, toutes les entrées déjà accédées sont aussi mises en cache en mémoire, et ré-écrites sur `sync()` et `close()` ; cela peut faciliter la modification des entrées modifiables dans le dictionnaire persistant, mais, si vous accédez à beaucoup d'entrées, cela peut consommer beaucoup de mémoire cache, et cela peut rendre l'opération de fermeture très lente puisque toutes les entrées déjà accédées sont ré-écrites (il n'y a aucun moyen de savoir quelles entrées déjà accédées sont mutables, ni lesquelles ont été vraiment modifiées).

Note : Ne pas se fier à la fermeture automatique de *shelf* ; appelez toujours `close()` explicitement quand vous n'en avez plus besoin, ou utilisez `shelve.open()` comme un gestionnaire de contexte :

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

Avertissement : Puisque le module *shelve* utilise en arrière plan *pickle*, il n'est pas sûr de charger un *shelf* depuis une source non fiable. Comme avec *pickle*, charger un *shelf* peut exécuter du code arbitraire.

Les objets *shelf* gèrent toutes les méthodes des dictionnaires. Cela facilite la transition depuis les scripts utilisant des dictionnaires à ceux nécessitant un stockage persistant.

Deux méthodes supplémentaires sont supportées :

`Shelf.sync()`

Réécrit toutes les entrées dans le cache si le *shelf* a été ouvert avec *writeback* passé à `True`. Vide le cache et synchronise le dictionnaire persistant sur le disque, si faisable. Elle est appelée automatiquement quand le *shelf* est fermé avec `close()`.

`Shelf.close()`

Synchronise et ferme l'objet *dict* persistant. Les opérations sur un *shelf* fermé échouent avec une *ValueError*.

Voir aussi :

[Recette pour un dictionnaire persistant](#) avec un large panel de formats de stockage et ayant la vitesse des dictionnaires natifs.

12.3.1 Limites

- Le choix du paquet de base de données à utiliser (comme `dbm.ndbm` ou `dbm.gnu`) dépend de l'interface disponible. Donc c'est risqué d'ouvrir la base de données directement en utilisant `dbm`. La base de données est également (malheureusement) sujette à des limitations de `dbm`, si c'est utilisé — cela signifie que (la représentation *pickled* de) l'objet stocké dans la base de données doit être assez petit et, dans de rare cas des collisions de clés peuvent entraîner le refus de mises à jour de la base de données.
- Le module `shelve` ne gère pas l'accès *concurrent* en lecture/écriture sur les objets stockés (les accès simultanés en lecture sont sûrs). Lorsqu'un programme a un *shelf* ouvert en écriture, aucun autre programme ne doit l'avoir ouvert en écriture ou lecture. Le verrouillage des fichiers Unix peut être utilisé pour résoudre ce problème, mais cela dépend de la version Unix et nécessite des connaissances à propos de l'implémentation de la base de données utilisée.

class `shelve.Shelf` (*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

Sous-classe de `collections.abc.MutableMapping` qui stocke les valeurs sérialisées par *pickle* dans l'objet *dict*.

Par défaut, les *pickles* de version 3 sont utilisés pour sérialiser les valeurs. La version du protocole *pickle* peut être spécifiée avec le paramètre *protocol*. Voir la documentation de *pickle* pour plus d'informations sur les protocoles *pickle*.

Si le paramètre *writeback* est `True`, l'objet garde en cache toutes les entrées accédées et les écrit dans le *dict* aux moments de synchronisation et de fermeture. Cela permet des opérations naturelles sur les entrées modifiables, mais peut consommer beaucoup plus de mémoire et rendre les temps de synchronisation et de fermeture très longs.

Le paramètre *keyencoding* est l'encodage utilisé pour encoder les clés avant qu'elles soient utilisées avec le dictionnaire sous-jacent.

Un objet *Shelf* peut également être utilisé comme un gestionnaire de contexte ; il est automatiquement fermé lorsque le bloc `with` est terminé.

Modifié dans la version 3.2 : Ajout du paramètre *keyencoding* ; précédemment, les clés étaient toujours encodées en UTF-8.

Modifié dans la version 3.4 : Ajout de la gestion des gestionnaires de contexte.

class `shelve.BsddbShelf` (*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

Sous-classe de *Shelf* qui propose `first()`, `next()`, `previous()`, `last()` et `set_location()` qui sont disponibles dans le module tiers `bsddb` de `pybsddb` mais non dans les autres modules de base de données. L'objet *dict* passé au constructeur doit savoir gérer ces méthodes. Cela est généralement fait en appelant une des fonctions suivantes : `bsddb.hashopen()`, `bsddb.btopen()` ou `bsddb.rnopen()`. Les paramètres optionnels *protocol*, *writeback*, et *keyencoding* ont la même signification que pour la classe *Shelf*.

class `shelve.DbfilenameShelf` (*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Sous-classe de *Shelf* qui accepte un *filename* au lieu d'un objet dictionnaire-compatible. Le fichier sous-jacent est ouvert avec `dbm.open()`. Par défaut le fichier est créé en lecture et en écriture. Le paramètre optionnel *flag* peut être interprété de la même manière que pour la fonction `open()`. Les paramètres optionnels *protocol* et *writeback* s'interprètent de la même manière que pour la classe *Shelf*.

12.3.2 Exemple

Pour résumer l'interface (key est une chaîne de caractère, data est un objet arbitraire) :

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

Voir aussi :

Module `dbm` Interface générique de base de données style dbm.

Module `pickle` Sérialisation d'objet utilisé par `shelve`.

12.4 `marshal` — Internal Python object serialization

This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python value to a file on a PC, transport the file to a Sun, and read it back there). Details of the format are undocumented on purpose ; it may change between Python versions (although it rarely does).¹

This is not a general « persistence » module. For general persistence and transfer of Python objects through RPC calls, see the modules `pickle` and `shelve`. The `marshal` module exists mainly to support reading and writing the « pseudo-compiled » code for Python modules of `.pyc` files. Therefore, the Python maintainers reserve the right to modify the marshal format in backward incompatible ways should the need arise. If you're serializing and de-serializing Python

1. The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who use the term « marshalling » for shipping of data around in a self-contained form. Strictly speaking, « to marshal » means to convert some data from internal to external form (in an RPC buffer for instance) and « unmarshalling » for the reverse process.

objects, use the `pickle` module instead – the performance is comparable, version independence is guaranteed, and pickle supports a substantially wider range of objects than marshal.

Avertissement : The `marshal` module is not intended to be secure against erroneous or maliciously constructed data. Never unmarshal data received from an untrusted or unauthenticated source.

Not all Python object types are supported ; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported : booleans, integers, floating point numbers, complex numbers, strings, bytes, bytearrays, tuples, lists, sets, frozensets, dictionaries, and code objects, where it should be understood that tuples, lists, sets, frozensets and dictionaries are only supported as long as the values contained therein are themselves supported. The singletons `None`, `Ellipsis` and `StopIteration` can also be marshalled and unmarshalled. For format *version* lower than 3, recursive lists, sets and dictionaries cannot be written (see below).

There are functions that read/write files as well as functions operating on bytes-like objects.

The module defines these functions :

`marshal.dump(value, file[, version])`

Write the value on the open file. The value must be a supported type. The file must be a writeable *binary file*.

If the value has (or contains an object that has) an unsupported type, a `ValueError` exception is raised — but garbage data will also be written to the file. The object will not be properly read back by `load()`.

The *version* argument indicates the data format that dump should use (see below).

`marshal.load(file)`

Read one value from the open file and return it. If no valid value is read (e.g. because the data has a different Python version's incompatible marshal format), raise `EOFError`, `ValueError` or `TypeError`. The file must be a readable *binary file*.

Note : If an object containing an unsupported type was marshalled with `dump()`, `load()` will substitute `None` for the unmarshallable type.

`marshal.dumps(value[, version])`

Return the bytes object that would be written to a file by `dump(value, file)`. The value must be a supported type. Raise a `ValueError` exception if value has (or contains an object that has) an unsupported type.

The *version* argument indicates the data format that dumps should use (see below).

`marshal.loads(bytes)`

Convert the *bytes-like object* to a value. If no valid value is found, raise `EOFError`, `ValueError` or `TypeError`. Extra bytes in the input are ignored.

In addition, the following constants are defined :

`marshal.version`

Indicates the format that the module uses. Version 0 is the historical format, version 1 shares interned strings and version 2 uses a binary format for floating point numbers. Version 3 adds support for object instancing and recursion. The current version is 4.

Notes

12.5 dbm — Interfaces to Unix « databases »

Source code : `Lib/dbm/__init__.py`

`dbm` is a generic interface to variants of the DBM database — `dbm.gnu` or `dbm.ndbm`. If none of these modules is installed, the slow-but-simple implementation in module `dbm.dumb` will be used. There is a [third party interface](#) to the Oracle Berkeley DB.

exception `dbm.error`

A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named `dbm.error` as the first item — the latter is used when `dbm.error` is raised.

`dbm.whichdb(filename)`

This function attempts to guess which of the several simple database modules available — `dbm.gnu`, `dbm.ndbm` or `dbm.dumb` — should be used to open a given file.

Returns one of the following values : `None` if the file can't be opened because it's unreadable or doesn't exist ; the empty string (`' '`) if the file's format can't be guessed ; or a string containing the required module name, such as `'dbm.ndbm'` or `'dbm.gnu'`.

`dbm.open(file, flag='r', mode=0o666)`

Open the database file `file` and return a corresponding object.

If the database file already exists, the `whichdb()` function is used to determine its type and the appropriate module is used ; if it does not exist, the first module listed above that can be imported is used.

The optional `flag` argument can be :

Valeur	Signification
'r'	Open existing database for reading only (default)
'w'	Open existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn't exist
'n'	Always create a new, empty database, open for reading and writing

The optional `mode` argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal `0o666` (and will be modified by the prevailing `umask`).

The object returned by `open()` supports the same basic functionality as dictionaries ; keys and their corresponding values can be stored, retrieved, and deleted, and the `in` operator and the `keys()` method are available, as well as `get()` and `setdefault()`.

Modifié dans la version 3.2 : `get()` and `setdefault()` are now available in all database modules.

Key and values are always stored as bytes. This means that when strings are used they are implicitly converted to the default encoding before being stored.

These objects also support being used in a `with` statement, which will automatically close them when done.

Modifié dans la version 3.4 : Added native support for the context management protocol to the objects returned by `open()`.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database :

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:
```

(suite sur la page suivante)

```
# Record some values
db[b'hello'] = b'there'
db['www.python.org'] = 'Python Website'
db['www.cnn.com'] = 'Cable News Network'

# Note that the keys are considered bytes now.
assert db[b'www.python.org'] == b'Python Website'
# Notice how the value is now in bytes.
assert db['www.cnn.com'] == b'Cable News Network'

# Often-used methods of the dict interface work too.
print(db.get('python.org', b'not present'))

# Storing a non-string key or value will raise an exception (most
# likely a TypeError).
db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

Voir aussi :

Module *shelve* Persistence module which stores non-string data.

The individual submodules are described in the following sections.

12.5.1 *dbm.gnu* — GNU’s reinterpretation of *dbm*

Source code : [Lib/dbm/gnu.py](#)

This module is quite similar to the *dbm* module, but uses the GNU library *gdbm* instead to provide some additional functionality. Please note that the file formats created by *dbm.gnu* and *dbm.ndbm* are incompatible.

The *dbm.gnu* module provides an interface to the GNU DBM library. *dbm.gnu.gdbm* objects behave like mappings (dictionaries), except that keys and values are always converted to bytes before storing. Printing a *gdbm* object doesn’t print the keys and values, and the *items()* and *values()* methods are not supported.

exception *dbm.gnu.error*

Raised on *dbm.gnu*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.gnu.open(*filename*[, *flag*[, *mode*]])

Open a *gdbm* database and return a *gdbm* object. The *filename* argument is the name of the database file.

The optional *flag* argument can be :

Valeur	Signification
'r'	Open existing database for reading only (default)
'w'	Open existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn’t exist
'n'	Always create a new, empty database, open for reading and writing

The following additional characters may be appended to the flag to control how the database is opened :

Valeur	Signification
'f'	Open the database in fast mode. Writes to the database will not be synchronized.
's'	Synchronized mode. This will cause changes to the database to be immediately written to the file.
'u'	Do not lock database.

Not all flags are valid for all versions of `gdbm`. The module constant `open_flags` is a string of supported flag characters. The exception `error` is raised if an invalid flag is specified.

The optional `mode` argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal `0o666`.

In addition to the dictionary-like methods, `gdbm` objects have the following methods :

`gdbm.firstkey()`

It's possible to loop over every key in the database using this method and the `nextkey()` method. The traversal is ordered by `gdbm`'s internal hash values, and won't be sorted by the key values. This method returns the starting key.

`gdbm.nextkey(key)`

Returns the key that follows `key` in the traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all :

```
k = db.firstkey()
while k != None:
    print(k)
    k = db.nextkey(k)
```

`gdbm.reorganize()`

If you have carried out a lot of deletions and would like to shrink the space used by the `gdbm` file, this routine will reorganize the database. `gdbm` objects will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

`gdbm.sync()`

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

`gdbm.close()`

Close the `gdbm` database.

12.5.2 `dbm.ndbm` — Interface based on `ndbm`

Source code : [Lib/dbm/ndbm.py](#)

The `dbm.ndbm` module provides an interface to the Unix « (n)dbm » library. Dbm objects behave like mappings (dictionaries), except that keys and values are always stored as bytes. Printing a `dbm` object doesn't print the keys and values, and the `items()` and `values()` methods are not supported.

This module can be used with the « classic » `ndbm` interface or the GNU GDBM compatibility interface. On Unix, the `configure` script will attempt to locate the appropriate header file to simplify building this module.

exception `dbm.ndbm.error`

Raised on `dbm.ndbm`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

`dbm.ndbm.library`

Name of the `ndbm` implementation library used.

`dbm.ndbm.open(filename[, flag[, mode]])`

Open a dbm database and return a `ndbm` object. The `filename` argument is the name of the database file (without the `.dir` or `.pag` extensions).

The optional *flag* argument must be one of these values :

Valeur	Signification
'r'	Open existing database for reading only (default)
'w'	Open existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn't exist
'n'	Always create a new, empty database, open for reading and writing

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0o666 (and will be modified by the prevailing umask).

In addition to the dictionary-like methods, `ndbm` objects provide the following method :

`ndbm.close()`

Close the `ndbm` database.

12.5.3 `dbm.dumb` — Portable DBM implementation

Source code : [Lib/dbm/dumb.py](#)

Note : The `dbm.dumb` module is intended as a last resort fallback for the `dbm` module when a more robust module is not available. The `dbm.dumb` module is not written for speed and is not nearly as heavily used as the other database modules.

The `dbm.dumb` module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as `dbm.gnu` no external library is required. As with other persistent mappings, the keys and values are always stored as bytes.

Le module définit :

exception `dbm.dumb.error`

Raised on `dbm.dumb`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

`dbm.dumb.open(filename[, flag[, mode]])`

Open a `dumbdbm` database and return a `dumbdbm` object. The *filename* argument is the basename of the database file (without any specific extensions). When a `dumbdbm` database is created, files with `.dat` and `.dir` extensions are created.

The optional *flag* argument supports only the semantics of 'c' and 'n' values. Other values will default to database being always opened for update, and will be created if it does not exist.

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0o666 (and will be modified by the prevailing umask).

Modifié dans la version 3.5 : `open()` always creates a new database when the flag has the value 'n'.

Deprecated since version 3.6, will be removed in version 3.8 : Creating database in 'r' and 'w' modes. Modifying database in 'r' mode.

In addition to the methods provided by the `collections.abc.MutableMapping` class, `dumbdbm` objects provide the following methods :

`dumbdbm.sync()`

Synchronize the on-disk directory and data files. This method is called by the `Shelve.sync()` method.

`dumbdbm.close()`

Close the `dumbdbm` database.

12.6 sqlite3 — Interface DB-API 2.0 pour bases de données SQLite

Code source : [Lib/sqlite3/](#)

SQLite est une bibliothèque C qui fournit une base de données légère sur disque ne nécessitant pas de processus serveur et qui utilise une variante (non standard) du langage de requête SQL pour accéder aux données. Certaines applications peuvent utiliser SQLite pour le stockage de données internes. Il est également possible de créer une application prototype utilisant SQLite, puis de modifier le code pour utiliser une base de données plus robuste telle que PostgreSQL ou Oracle.

Le module *sqlite3* a été écrit par Gerhard Häring. Il fournit une interface SQL conforme à la spécification DB-API 2.0 décrite par [PEP 249](#).

Pour utiliser le module, vous devez d'abord créer une *Connection* qui représente la base de données. Dans cet exemple, les données sont stockées dans le fichier `example.db` :

```
import sqlite3
conn = sqlite3.connect('example.db')
```

Vous pouvez également fournir le nom spécial `:memory:` pour créer une base de données dans la mémoire vive.

Une fois que vous avez une instance de *Connection*, vous pouvez créer un objet *Cursor* et appeler sa méthode *execute()* pour exécuter les commandes SQL :

```
c = conn.cursor()

# Create table
c.execute('CREATE TABLE stocks
          (date text, trans text, symbol text, qty real, price real)')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

Les données que vous avez sauvegardées sont persistantes et disponibles dans les sessions suivantes :

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```

Habituellement, vos opérations SQL utilisent les valeurs de variables Python. Vous ne devez pas assembler votre requête à l'aide des opérations sur les chaînes de caractères de Python, car cela n'est pas sûr. Cela rend votre programme vulnérable à une attaque par injection SQL (voir <https://xkcd.com/327/> pour un exemple amusant de ce qui peut mal tourner).

À la place, utilisez la capacité DB-API de substitution des paramètres. Placez un `?` comme indicateur partout où vous voulez utiliser une valeur, puis fournissez un *tuple* de valeurs comme second argument de la méthode *execute()*. D'autres modules de base de données peuvent utiliser un espace réservé différent, tel que `%s` ou `:1`. Par exemple :

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print(c.fetchone())

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
              ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
              ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
              ]
c.executemany('INSERT INTO stocks VALUES (?, ?, ?, ?, ?)', purchases)
```

Pour récupérer des données après avoir exécuté une instruction *SELECT*, vous pouvez considérer le curseur comme un *itérateur*, appeler la méthode du curseur *fetchone()* pour récupérer une seule ligne correspondante ou appeler *fetchall()* pour obtenir une liste des lignes correspondantes.

Cet exemple utilise la forme itérateur :

```
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
        print(row)

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

Voir aussi :

<https://github.com/ghaering/pysqlite> La page web de *pysqlite* — *sqlite3* est développée sur un site tiers sous le nom *pysqlite*.

<https://www.sqlite.org> Dans la page Web de SQLite, la documentation décrit la syntaxe et les types de données disponibles qui sont pris en charge par cette variante SQL.

<http://www.w3schools.com/sql/> Tutoriel, référence et exemples pour apprendre la syntaxe SQL.

PEP 249 — Spécifications de l'API 2.0 pour la base de données PEP écrite par Marc-André Lemburg.

12.6.1 Fonctions et constantes du module

`sqlite3.version`

Le numéro de version de ce module, sous forme de chaîne. Ce n'est pas la version de la bibliothèque SQLite.

`sqlite3.version_info`

Le numéro de version de ce module, sous forme d'un n-uplet d'entiers. Ce n'est pas la version de la bibliothèque SQLite.

`sqlite3.sqlite_version`

Le numéro de version de la bibliothèque d'exécution SQLite, sous forme de chaîne.

`sqlite3.sqlite_version_info`

Le numéro de version de la bibliothèque d'exécution SQLite, sous forme d'entier.

`sqlite3.PARSE_DECLTYPES`

Cette constante est destinée à être utilisée avec le paramètre *detect_types* de la fonction *connect()*.

Si elle est définie, le module *sqlite3* analyse le type de donnée déclarée pour chaque colonne. Il déduit le type du premier mot de la déclaration, par exemple de *integer primary key* il gardera *integer*, ou de *number(10)* il gardera *number*. Ensuite, pour cette colonne, il utilisera une fonction de conversion du dictionnaire des convertisseurs.

sqlite3.PARSE_COLNAMES

Cette constante est destinée à être utilisée avec le paramètre *detect_types* de la fonction *connect()*.

Permet à l'interface SQLite d'analyser le nom pour chaque colonne. Il y cherchera une chaîne comme `[mytype]` indiquant que la colonne est de type *mytype*. Il essaiera de trouver une entrée *mytype* dans le dictionnaire, puis utilisera la fonction de conversion qui s'y trouve pour renvoyer la valeur. Le nom de colonne donnée à *Cursor.description* n'est alors que le premier mot du nom de la colonne, par exemple si vous utilisez `'as \'x [datetime]\'` dans votre code SQL, le nom de la colonne sera simplement *x*.

sqlite3.connect(*database*[, *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, *uri*])

Opens a connection to the SQLite database file *database*. You can use `":memory:"` to open a database connection to a database that resides in RAM instead of on disk.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The *timeout* parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).

For the *isolation_level* parameter, please see the *isolation_level* property of *Connection* objects.

SQLite natively supports only the types TEXT, INTEGER, REAL, BLOB and NULL. If you want to use other types you must add support for them yourself. The *detect_types* parameter and the using custom **converters** registered with the module-level *register_converter()* function allow you to easily do that.

detect_types defaults to 0 (i. e. off, no type detection), you can set it to any combination of *PARSE_DECLTYPES* and *PARSE_COLNAMES* to turn type detection on.

By default, *check_same_thread* is *True* and only the creating thread may use the connection. If set *False*, the returned connection may be shared across multiple threads. When using multiple threads with the same connection writing operations should be serialized by the user to avoid data corruption.

By default, the *sqlite3* module uses its *Connection* class for the connect call. You can, however, subclass the *Connection* class and make *connect()* use your class instead by providing your class for the *factory* parameter.

Consult the section *SQLite and Python types* of this manual for details.

The *sqlite3* module internally uses a statement cache to avoid SQL parsing overhead. If you want to explicitly set the number of statements that are cached for the connection, you can set the *cached_statements* parameter. The currently implemented default is to cache 100 statements.

If *uri* is true, *database* is interpreted as a URI. This allows you to specify options. For example, to open a database in read-only mode you can use :

```
db = sqlite3.connect('file:path/to/database?mode=ro', uri=True)
```

More information about this feature, including a list of recognized options, can be found in the [SQLite URI documentation](#).

Modifié dans la version 3.4 : Added the *uri* parameter.

sqlite3.register_converter(*typename*, *callable*)

Registers a callable to convert a bytestring from the database into a custom Python type. The callable will be invoked for all database values that are of the type *typename*. Confer the parameter *detect_types* of the *connect()* function for how the type detection works. Note that *typename* and the name of the type in your query are matched in case-insensitive manner.

sqlite3.register_adapter(*type*, *callable*)

Registers a callable to convert the custom Python type *type* into one of SQLite's supported types. The callable *callable* accepts as single parameter the Python value, and must return a value of the following types : int, float, str or bytes.

sqlite3.complete_statement(*sql*)

Returns *True* if the string *sql* contains one or more complete SQL statements terminated by semicolons. It does not verify that the SQL is syntactically correct, only that there are no unclosed string literals and the statement is terminated by a semicolon.

This can be used to build a shell for SQLite, as in the following example :

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print("Enter your SQL commands to execute in sqlite3.")
print("Enter a blank line to exit.")

while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print(cur.fetchall())
        except sqlite3.Error as e:
            print("An error occurred:", e.args[0])
        buffer = ""

con.close()
```

`sqlite3.enable_callback_tracebacks(flag)`

By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with *flag* set to `True`. Afterwards, you will get tracebacks from callbacks on `sys.stderr`. Use `False` to disable the feature again.

12.6.2 Objets de connexions

class `sqlite3.Connection`

A SQLite database connection has the following attributes and methods :

isolation_level

Get or set the current default isolation level. `None` for autocommit mode or one of « DEFERRED », « IMMEDIATE » or « EXCLUSIVE ». See section [Controlling Transactions](#) for a more detailed explanation.

in_transaction

`True` if a transaction is active (there are uncommitted changes), `False` otherwise. Read-only attribute.

Nouveau dans la version 3.2.

cursor (*factory=Cursor*)

The cursor method accepts a single optional parameter *factory*. If supplied, this must be a callable returning an instance of `Cursor` or its subclasses.

commit ()

This method commits the current transaction. If you don't call this method, anything you did since the last call to `commit` () is not visible from other database connections. If you wonder why you don't see the data you've written to the database, please check you didn't forget to call this method.

rollback()

This method rolls back any changes to the database since the last call to `commit()`.

close()

This closes the database connection. Note that this does not automatically call `commit()`. If you just close your database connection without calling `commit()` first, your changes will be lost!

execute(sql[, parameters])

This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor's `execute()` method with the `parameters` given, and returns the cursor.

executemany(sql[, parameters])

This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor's `executemany()` method with the `parameters` given, and returns the cursor.

executescript(sql_script)

This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor's `executescript()` method with the given `sql_script`, and returns the cursor.

create_function(name, num_params, func)

Creates a user-defined function that you can later use from within SQL statements under the function name `name`. `num_params` is the number of parameters the function accepts (if `num_params` is -1, the function may take any number of arguments), and `func` is a Python callable that is called as the SQL function.

The function can return any of the types supported by SQLite : bytes, str, int, float and None.

Exemple :

```
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", (b"foo",))
print(cur.fetchone()[0])
```

create_aggregate(name, num_params, aggregate_class)

Creates a user-defined aggregate function.

The aggregate class must implement a `step` method, which accepts the number of parameters `num_params` (if `num_params` is -1, the function may take any number of arguments), and a `finalize` method which will return the final result of the aggregate.

The `finalize` method can return any of the types supported by SQLite : bytes, str, int, float and None.

Exemple :

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
```

(suite sur la page suivante)

(suite de la page précédente)

```

cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print(cur.fetchone()[0])

```

create_collation (*name*, *callable*)

Creates a collation with the specified *name* and *callable*. The callable will be passed two string arguments. It should return -1 if the first is ordered lower than the second, 0 if they are ordered equal and 1 if the first is ordered higher than the second. Note that this controls sorting (ORDER BY in SQL) so your comparisons don't affect other SQL operations.

Note that the callable will get its parameters as Python bytestrings, which will normally be encoded in UTF-8. The following example shows a custom collation that sorts « the wrong way » :

```

import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print(row)
con.close()

```

To remove a collation, call `create_collation` with `None` as callable :

```
con.create_collation("reverse", None)
```

interrupt ()

You can call this method from a different thread to abort any queries that might be executing on the connection. The query will then abort and the caller will get an exception.

set_authorizer (*authorizer_callback*)

This routine registers a callback. The callback is invoked for each attempt to access a column of a table in the database. The callback should return `SQLITE_OK` if access is allowed, `SQLITE_DENY` if the entire SQL statement should be aborted with an error and `SQLITE_IGNORE` if the column should be treated as a NULL value. These constants are available in the `sqlite3` module.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or `None` depending on the first argument. The 4th argument is the name of the database (« main », « temp », etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or `None` if this access attempt is directly from input SQL code. Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the `sqlite3` module.

set_progress_handler (*handler*, *n*)

This routine registers a callback. The callback is invoked for every *n* instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with *None* for *handler*.

Returning a non-zero value from the handler function will terminate the currently executing query and cause it to raise an *OperationalError* exception.

set_trace_callback (*trace_callback*)

Registers *trace_callback* to be called for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as string) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the *Cursor.execute()* methods. Other sources include the transaction management of the Python module and the execution of triggers defined in the current database.

Passing *None* as *trace_callback* will disable the trace callback.

Nouveau dans la version 3.3.

enable_load_extension (*enabled*)

This routine allows/disallows the SQLite engine to load SQLite extensions from shared libraries. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

Loadable extensions are disabled by default. See ¹.

Nouveau dans la version 3.2.

```
import sqlite3

con = sqlite3.connect(":memory:")

# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli
↪peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin
↪onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli
↪cheese onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin
↪sugar flour butter');
""")
for row in con.execute("select rowid, name, ingredients from recipe where
↪name match 'pie'"):
    print(row)
```

1. The *sqlite3* module is not built with loadable extension support by default, because some platforms (notably Mac OS X) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass `--enable-loadable-sqlite-extensions` to configure.

load_extension (*path*)

This routine loads a SQLite extension from a shared library. You have to enable extension loading with `enable_load_extension()` before you can use this routine.

Loadable extensions are disabled by default. See¹.

Nouveau dans la version 3.2.

row_factory

You can change this attribute to a callable that accepts the cursor and the original row as a tuple and will return the real result row. This way, you can implement more advanced ways of returning results, such as returning an object that can also access columns by name.

Exemple :

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone() ["a"])
```

If returning a tuple doesn't suffice and you want name-based access to columns, you should consider setting `row_factory` to the highly-optimized `sqlite3.Row` type. `Row` provides both index-based and case-insensitive name-based access to columns with almost no memory overhead. It will probably be better than your own custom dictionary-based approach or even a `db_row` based solution.

text_factory

Using this attribute you can control what objects are returned for the TEXT data type. By default, this attribute is set to `str` and the `sqlite3` module will return Unicode objects for TEXT. If you want to return bytestrings instead, you can set it to `bytes`.

You can also set it to any other callable that accepts a single bytestring parameter and returns the resulting object.

See the following example code for illustration :

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = "\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
```

(suite sur la page suivante)

(suite de la page précédente)

```

assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8") + "foo"
cur.execute("select ?", ("bar",))
row = cur.fetchone()
assert row[0] == "barfoo"

```

total_changes

Returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

iterdump()

Returns an iterator to dump the database in an SQL text format. Useful when saving an in-memory database for later restoration. This function provides the same capabilities as the `.dump` command in the **sqlite3** shell.

Exemple :

```

# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)

```

12.6.3 Cursor Objects

class sqlite3.Cursor

A *Cursor* instance has the following attributes and methods.

execute (sql[, parameters])

Executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The *sqlite3* module supports two kinds of placeholders : question marks (qmark style) and named placeholders (named style).

Here's an example of both styles :

```

import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table people (name_last, age)")

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age", {"who": who, "age": age})

print (cur.fetchone())

```

`execute()` will only execute a single SQL statement. If you try to execute more than one statement with it, it will raise a *Warning*. Use `executescript()` if you want to execute multiple SQL statements with one call.

executemany (*sql*, *seq_of_parameters*)

Executes an SQL command against all parameter sequences or mappings found in the sequence *seq_of_parameters*. The *sqlite3* module also allows using an *iterator* yielding parameters instead of a sequence.

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print(cur.fetchall())
```

Here's a shorter example using a *generator* :

```
import sqlite3
import string

def char_generator():
    for c in string.ascii_lowercase:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print(cur.fetchall())
```

executescript (*sql_script*)

This is a nonstandard convenience method for executing multiple SQL statements at once. It issues a `COMMIT` statement first, then executes the SQL script it gets as a parameter.

sql_script can be an instance of *str*.

Exemple :

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently's Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
""")
```

fetchone()

Fetches the next row of a query result set, returning a single sequence, or *None* when no more data is available.

fetchmany(size=cursor.arraysize)

Fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available.

The number of rows to fetch per call is specified by the *size* parameter. If it is not given, the cursor's *arraysize* determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the *size* parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.

Note there are performance considerations involved with the *size* parameter. For optimal performance, it is usually best to use the *arraysize* attribute. If the *size* parameter is used, then it is best for it to retain the same value from one *fetchmany()* call to the next.

fetchall()

Fetches all (remaining) rows of a query result, returning a list. Note that the cursor's *arraysize* attribute can affect the performance of this operation. An empty list is returned when no rows are available.

close()

Close the cursor now (rather than whenever `__del__` is called).

The cursor will be unusable from this point forward; a *ProgrammingError* exception will be raised if any operation is attempted with the cursor.

rowcount

Although the *Cursor* class of the *sqlite3* module implements this attribute, the database engine's own support for the determination of « rows affected »/ « rows selected » is quirky.

For *executemany()* statements, the number of modifications are summed up into *rowcount*.

As required by the Python DB API Spec, the *rowcount* attribute « is -1 in case no *executeXX()* has been performed on the cursor or the rowcount of the last operation is not determinable by the interface ». This includes *SELECT* statements because we cannot determine the number of rows a query produced until all rows were fetched.

With SQLite versions before 3.6.5, *rowcount* is set to 0 if you make a *DELETE FROM table* without any condition.

lastrowid

This read-only attribute provides the rowid of the last modified row. It is only set if you issued an `INSERT` or a `REPLACE` statement using the `execute()` method. For operations other than `INSERT` or `REPLACE` or when `executemany()` is called, `lastrowid` is set to `None`.

If the `INSERT` or `REPLACE` statement failed to insert the previous successful rowid is returned.

Modifié dans la version 3.6 : Added support for the `REPLACE` statement.

arraysize

Read/write attribute that controls the number of rows returned by `fetchmany()`. The default value is 1 which means a single row would be fetched per call.

description

This read-only attribute provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are `None`.

It is set for `SELECT` statements without any matching rows as well.

connection

This read-only attribute provides the SQLite database `Connection` used by the `Cursor` object. A `Cursor` object created by calling `con.cursor()` will have a `connection` attribute that refers to `con` :

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

12.6.4 Row Objects

class `sqlite3.Row`

A `Row` instance serves as a highly optimized `row_factory` for `Connection` objects. It tries to mimic a tuple in most of its features.

It supports mapping access by column name and index, iteration, representation, equality testing and `len()`.

If two `Row` objects have exactly the same columns and their members are equal, they compare equal.

keys()

This method returns a list of column names. Immediately after a query, it is the first member of each tuple in `Cursor.description`.

Modifié dans la version 3.5 : Added support of slicing.

Let's assume we initialize a table as in the example given above :

```
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute('''create table stocks
(date text, trans text, symbol text,
qty real, price real)''')
c.execute("""insert into stocks
values ('2006-01-05','BUY','RHAT',100,35.14) """)
conn.commit()
c.close()
```

Now we plug `Row` in :

```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
```

(suite sur la page suivante)

(suite de la page précédente)

```

>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14

```

12.6.5 Exceptions

exception `sqlite3.Warning`

A subclass of *Exception*.

exception `sqlite3.Error`

The base class of the other exceptions in this module. It is a subclass of *Exception*.

exception `sqlite3.DatabaseError`

Exception raised for errors that are related to the database.

exception `sqlite3.IntegrityError`

Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails. It is a subclass of *DatabaseError*.

exception `sqlite3.ProgrammingError`

Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc. It is a subclass of *DatabaseError*.

exception `sqlite3.OperationalError`

Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, etc. It is a subclass of *DatabaseError*.

exception `sqlite3.NotSupportedError`

Exception raised in case a method or database API was used which is not supported by the database, e.g. calling the *rollback()* method on a connection that does not support transaction or has transactions turned off. It is a subclass of *DatabaseError*.

12.6.6 SQLite and Python types

Introduction

SQLite natively supports the following types : NULL, INTEGER, REAL, TEXT, BLOB.

The following Python types can thus be sent to SQLite without any problem :

Type Python	SQLite type
<i>None</i>	NULL
<i>int</i>	INTEGER
<i>float</i>	REAL
<i>str</i>	TEXT
<i>bytes</i>	BLOB

This is how SQLite types are converted to Python types by default :

SQLite type	Type Python
NULL	<i>None</i>
INTEGER	<i>int</i>
REAL	<i>float</i>
TEXT	depends on <i>text_factory</i> , <i>str</i> by default
BLOB	<i>bytes</i>

The type system of the *sqlite3* module is extensible in two ways : you can store additional Python types in a SQLite database via object adaptation, and you can let the *sqlite3* module convert SQLite types to different Python types via converters.

Using adapters to store additional Python types in SQLite databases

As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must **adapt** them to one of the *sqlite3* module's supported types for SQLite : one of `NoneType`, `int`, `float`, `str`, `bytes`.

There are two ways to enable the *sqlite3* module to adapt a custom Python type to one of the supported ones.

Letting your object adapt itself

This is a good approach if you write the class yourself. Let's suppose you have a class like this :

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

Now you want to store the point in a single SQLite column. First you'll have to choose one of the supported types first to be used for representing the point. Let's just use `str` and separate the coordinates using a semicolon. Then you need to give your class a method `__conform__(self, protocol)` which must return the converted value. The parameter *protocol* will be `PrepareProtocol`.

```
import sqlite3

class Point:
    def __init__(self, x, y):
```

(suite sur la page suivante)

(suite de la page précédente)

```

        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

```

Registering an adapter callable

The other possibility is to create a function that converts the type to the string representation and register the function with `register_adapter()`.

```

import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

```

The `sqlite3` module has two default adapters for Python's built-in `datetime.date` and `datetime.datetime` types. Now let's suppose we want to store `datetime.datetime` objects not in ISO representation, but as a Unix timestamp.

```

import sqlite3
import datetime
import time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])

```

Converting SQLite values to custom Python types

Writing an adapter lets you send custom Python types to SQLite. But to make it really useful we need to make the Python to SQLite to Python roundtrip work.

Enter converters.

Let's go back to the `Point` class. We stored the `x` and `y` coordinates separated via semicolons as strings in SQLite.

First, we'll define a converter function that accepts the string as a parameter and constructs a `Point` object from it.

Note : Converter functions **always** get called with a *bytes* object, no matter under which data type you sent the value to SQLite.

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

Now you need to make the `sqlite3` module know that what you select from the database is actually a point. There are two ways of doing this :

- Implicitly via the declared type
- Explicitly via the column name

Both ways are described in section *Fonctions et constantes du module*, in the entries for the constants `PARSE_DECLTYPES` and `PARSE_COLNAMES`.

The following example illustrates both approaches.

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "({};{})".format(self.x, self.y)

def adapt_point(point):
    return "({};{})".format(point.x, point.y).encode('ascii')

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

#####
# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")
```

(suite sur la page suivante)

(suite de la page précédente)

```

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

#####
# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()

```

Default adapters and converters

There are default adapters for the date and datetime types in the datetime module. They will be sent as ISO dates/ISO timestamps to SQLite.

The default converters are registered under the name « date » for *datetime.date* and under the name « timestamp » for *datetime.datetime*.

This way, you can use date/timestamps from Python without any additional fiddling in most cases. The format of the adapters is also compatible with the experimental SQLite date/time functions.

The following example demonstrates this.

```

import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_
↳COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"
↳')
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
print("current_timestamp", row[1], type(row[1]))

```

If a timestamp stored in SQLite has a fractional part longer than 6 numbers, its value will be truncated to microsecond precision by the timestamp converter.

12.6.7 Controlling Transactions

The underlying `sqlite3` library operates in `autocommit` mode by default, but the Python `sqlite3` module by default does not.

`autocommit` mode means that statements that modify the database take effect immediately. A `BEGIN` or `SAVEPOINT` statement disables `autocommit` mode, and a `COMMIT`, a `ROLLBACK`, or a `RELEASE` that ends the outermost transaction, turns `autocommit` mode back on.

The Python `sqlite3` module by default issues a `BEGIN` statement implicitly before a Data Modification Language (DML) statement (i.e. `INSERT/UPDATE/DELETE/REPLACE`).

You can control which kind of `BEGIN` statements `sqlite3` implicitly executes via the `isolation_level` parameter to the `connect()` call, or via the `isolation_level` property of connections. If you specify no `isolation_level`, a plain `BEGIN` is used, which is equivalent to specifying `DEFERRED`. Other possible values are `IMMEDIATE` and `EXCLUSIVE`.

You can disable the `sqlite3` module's implicit transaction management by setting `isolation_level` to `None`. This will leave the underlying `sqlite3` library operating in `autocommit` mode. You can then completely control the transaction state by explicitly issuing `BEGIN`, `ROLLBACK`, `SAVEPOINT`, and `RELEASE` statements in your code.

Modifié dans la version 3.6 : `sqlite3` used to implicitly commit an open transaction before DDL statements. This is no longer the case.

12.6.8 Using `sqlite3` efficiently

Using shortcut methods

Using the nonstandard `execute()`, `executemany()` and `executescript()` methods of the `Connection` object, your code can be written more concisely because you don't have to create the (often superfluous) `Cursor` objects explicitly. Instead, the `Cursor` objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a `SELECT` statement and iterate over it directly using only a single call on the `Connection` object.

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)

# Print the table contents
for row in con.execute("select firstname, lastname from person"):
    print(row)

print("I just deleted", con.execute("delete from person").rowcount, "rows")
```

Accessing columns by name instead of by index

One useful feature of the `sqlite3` module is the built-in `sqlite3.Row` class designed to be used as a row factory.

Rows wrapped with this class can be accessed both by index (like tuples) and case-insensitively by name :

```
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

Using the connection as a context manager

Connection objects can be used as context managers that automatically commit or rollback transactions. In the event of an exception, the transaction is rolled back ; otherwise, the transaction is committed :

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# con.rollback() is called after the with block finishes with an exception, the
# exception is still raised and must be caught
try:
    with con:
        con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print("couldn't add Joe twice")
```

12.6.9 Common issues

Fils d'exécution

Older SQLite versions had issues with sharing connections between threads. That's why the Python module disallows sharing connections and cursors between threads. If you still try to do so, you will get an exception at runtime.

The only exception is calling the `interrupt()` method, which only makes sense to call from a different thread.

Notes

Compression de donnée et archivage

Les modules documentés dans ce chapitre implémentent les algorithmes de compression *zlib*, *gzip*, *bzip2* et *lzma*, ainsi que la création d'archives *ZIP* et *tar*. Voir aussi *Archiving operations* fourni par le module *shutil*.

13.1 *zlib* — Compression compatible avec *gzip*

Pour des applications nécessitant de compresser des données, les fonctions de ce module permettent la compression et la décompression via la bibliothèque *zlib*. La bibliothèque *zlib* a sa propre page web sur <http://www.zlib.net>. Il existe des incompatibilités connues entre le module Python et les versions de la bibliothèque *zlib* plus anciennes que la 1.1.3 ; 1.1.3 contient des failles de sécurité et nous recommandons d'utiliser plutôt la version 1.14 ou plus récente.

Les fonctions *zlib* recèlent de nombreuses options et il est nécessaire de suivre un ordre précis. Cette documentation n'a pas pour but de couvrir la globalité des possibilités. Aussi, veuillez consulter le manuel *zlib* en ligne sur <http://www.zlib.net/manual.html> pour compléter davantage son utilisation.

Pour lire ou écrire des fichiers *.gz* veuillez consulter le module *gzip*.

Les exceptions et fonctions disponibles dans ce module sont :

exception *zlib.error*

Exception levée lors d'erreurs de compression et de décompression.

***zlib.adler32* (*data* [, *value*])**

Calcule une somme de contrôle Adler-32 de *data* (une somme de contrôle Adler-32 est aussi fiable qu'un CRC32 mais peut être calculée bien plus rapidement). Le résultat produit est un entier non signé de 32-bit. Si *value* est défini, il devient la valeur initiale de la somme de contrôle ; sinon une valeur par défaut de 1 est utilisée. Définir *value* permet de calculer une somme de contrôle continue pendant la concaténation de plusieurs entrées. Cet algorithme n'a aucune garantie cryptographique puissante, et ne doit pas être utilisé ni pour l'authentification, ni pour des signatures numériques. Conçu comme un algorithme de somme de contrôle, il n'est pas adapté pour une utilisation sous forme de clé de hachage générique.

Modifié dans la version 3.0 : Renvoie une valeur non-signée. Pour produire la même valeur avec toutes les versions de Python sur différentes plateformes, utilisez `adler32(data) & 0xffffffff`.

`zlib.compress(data, level=-1)`

Comprime les octets contenus dans *data*, renvoie un objet *bytes* contenant les données compressées. *level* permet d'ajuster le niveau de compression, ce doit être un nombre entier compris entre 0 et 9 ou -1 ; 1 étant plus rapide et procède à une compression légère, 9 est plus lent mais compresses plus fortement. 0 n'effectue aucune compression. La valeur par défaut est -1 (`Z_DEFAULT_COMPRESSION`). `Z_DEFAULT_COMPRESSION` donne une valeur par défaut proposant un équilibre entre vitesse et taux de compression (actuellement équivalente à 6). Si une erreur surgit, l'exception `error` est levée.

Modifié dans la version 3.6 : *level* peut maintenant être passé par son nom.

`zlib.compressobj(level=-1, method=DEFLATED, wbits=MAX_WBITS, memLevel=DEF_MEM_LEVEL, strategy=Z_DEFAULT_STRATEGY[, zdict])`

Renvoie un objet « compresseur », à utiliser pour compresser des flux de données qui ne peuvent pas être stockés entièrement en mémoire.

level est le taux de compression – un entier compris entre 0 et 9 ou -1. La valeur 1 est la plus rapide avec taux de compression minimal, tandis que la valeur 9 est la plus lente mais produit une compression maximale. 0 ne produit aucune compression. La valeur par défaut est -1 (`Z_DEFAULT_COMPRESSION`). La constante `Z_DEFAULT_COMPRESSION` équivaut à un bon compromis par défaut entre rapidité et bonne compression (valeur équivalente au niveau 6).

method définit l'algorithme de compression. Pour le moment, la seule valeur acceptée est `DEFLATED`.

L'argument *wbits* contrôle la taille du tampon d'historique (« *window size* ») utilisé lors de la compression, et si un en-tête et un bloc final seront inclus. Il accepte plusieurs intervalles de valeurs, et vaut 15 (`MAX_WBITS`) par défaut :

- De +9 à +15 : le logarithme binaire de la taille du tampon, par conséquent compris entre 512 et 32 768. Des valeurs plus grandes produisent de meilleures compressions aux dépens d'une utilisation mémoire plus grande. Le résultat final inclus des en-têtes et des blocs spécifiques à *zlib*.
- De -9 à -15 : utilise la valeur absolue de *wbits* comme logarithme binaire de la taille du tampon, et ne produit pas d'en-têtes ni de bloc final.
- De +25 à +31 = 16 + (9 à 15) : utilise les 4 bits de poids faible comme logarithme binaire de la taille du tampon, tout en incluant un en-tête **gzip** et une somme de contrôle finale.

L'argument *memLevel* permet d'ajuster la quantité de mémoire utilisée pour stocker l'état interne de la compression. Les valeurs valides sont comprises entre 1 et 9. Des valeurs plus élevées occupent davantage de mémoire, mais sont plus rapides et produisent des sorties plus compressées.

strategy permet d'ajuster l'algorithme de compression. Les valeurs possibles sont `Z_DEFAULT_STRATEGY`, `Z_FILTERED`, et `Z_HUFFMAN_ONLY`, `Z_RLE` (*zlib* 1.2.0.1) et `Z_FIXED` (*zlib* 1.2.2.2).

zdict est un dictionnaire de compression prédéfini. C'est une séquence d'octets (tel qu'un objet *bytes*) contenant des sous-séquences attendues régulièrement dans les données à compresser. Les sous-séquences les plus fréquentes sont à placer à la fin du dictionnaire.

Modifié dans la version 3.3 : Ajout du paramètre *zdict*.

`zlib.crc32(data[, value])`

Calcule la somme de contrôle CRC (*Cyclic Redundancy Check* en anglais) de l'argument *data*. Il renvoie un entier non signé de 32 bits. Si l'argument *value* est présent, il permet de définir la valeur de départ de la somme de contrôle. Sinon, la valeur par défaut est 0. L'argument *value* permet de calculer la somme de contrôle glissante d'une concaténation de données. L'algorithme n'est pas fort d'un point de vue cryptographique, et ne doit pas être utilisé pour l'authentification ou des signatures numériques. Cet algorithme a été conçu pour être exploité comme un algorithme de somme de contrôle, ce n'est pas un algorithme de hachage générique.

Modifié dans la version 3.0 : Renvoie une valeur non-signée. Pour obtenir la même valeur sur n'importe quelle version de Python et n'importe quelle plateforme, utilisez `crc32(data) & 0xffffffff`.

`zlib.decompress(data, wbits=MAX_WBITS, bufsize=DEF_BUF_SIZE)`

Décompresses les octets de *data*, renvoyant un objet *bytes* contenant les données décompressées. Le paramètre *wbits* dépend du format des données compressées, et est abordé plus loin. Si l'argument *bufsize* est défini, il est utilisé comme taille initiale du tampon de sortie. En cas d'erreur, l'exception `error` est levée.

L'argument *wbits* contrôle la taille du tampon d'historique (« *window size* ») utilisé lors de la compression, et si un en-tête et un bloc final sont attendus. Similaire au paramètre de `compressobj()`, mais accepte une gamme plus

large de valeurs :

- De +8 à +15 : logarithme binaire pour la taille du tampon. L'entrée doit contenir un en-tête et un bloc *zlib*.
- 0 : détermine automatiquement la taille du tampon à partir de l'en-tête *zlib*. Géré uniquement depuis *zlib* 1.2.3.5.
- De -8 à -15 : utilise la valeur absolue de *wbits* comme logarithme binaire pour la taille du tampon. L'entrée doit être un flux brut, sans en-tête ni bloc final.
- De +24 à +31 = 16 + (8 à 15) : utilise les 4 de poids faible comme logarithme binaire pour la taille du tampon. L'entrée doit contenir un en-tête *gzip* et son bloc final.
- De +40 à +47 = 32 + (8 à 15) : utilise les 4 bits de poids faible comme logarithme binaire pour la taille du tampon, et accepte automatiquement les formats *zlib* ou *gzip*.

Lors de la décompression d'un flux, la taille du tampon ne doit pas être inférieure à la taille initialement utilisée pour compresser le flux. L'utilisation d'une valeur trop petite peut déclencher une exception *error*. La valeur par défaut *wbits* correspond à une taille élevée du tampon et nécessite d'y adjoindre un en-tête *zlib* et son bloc final.

L'argument *bufsize* correspond à la taille initiale du tampon utilisé pour contenir les données décompressées. Si plus d'espace est nécessaire, la taille du tampon sera augmentée au besoin, donc vous n'avez pas besoin de deviner la valeur exacte. Un réglage précis n'économisera que quelques appels à `malloc()`.

Modifié dans la version 3.6 : *wbits* et *bufsize* peuvent être utilisés comme arguments nommés.

`zlib.decompressobj(wbits=MAX_WBITS[, zdict])`

Renvoie un objet « décompresseur », à utiliser pour décompresser des flux de données qui ne rentrent pas entièrement en mémoire.

Le paramètre *wbits* contrôle la taille du tampon, et détermine quel format d'en-tête et de bloc sont prévus. Il a la même signification que décrit pour *decompress()*.

Le paramètre *zdict* définit un dictionnaire de compression prédéfini. S'il est fourni, il doit être identique au dictionnaire utilisé par le compresseur, à l'origine des données à décompresser.

Note : Si *zdict* est un objet modifiable (tel qu'un *bytearray*, par exemple), vous ne devez pas modifier son contenu entre l'appel à la fonction *decompressobj()* et le premier appel à la méthode *decompress()* du décompresseur.

Modifié dans la version 3.3 : Ajout du paramètre *zdict*.

Les objets de compression gèrent les méthodes suivantes :

`Compress.compress(data)`

Comprime *data* et renvoie au moins une partie des données compressées sous forme d'objet *bytes*. Ces données doivent être concaténées à la suite des appels précédant à *compress()*. Certaines entrées peuvent être conservées dans des tampons internes pour un traitement ultérieur.

`Compress.flush([mode])`

Toutes les entrées mises en attente sont traitées et un objet *bytes* contenant la sortie des données compressées restantes est renvoyé. L'argument *mode* accepte l'une des constantes suivantes : *Z_NO_FLUSH*, *Z_PARTIAL_FLUSH*, *Z_SYNC_FLUSH*, *Z_FULL_FLUSH*, *Z_BLOCK* (*zlib* 1.2.3.4), et *Z_FINISH*, par défaut *Z_FINISH*. Sauf *Z_FINISH*, toutes les constantes permettent de compresser d'autres chaînes d'octets, tandis que *Z_FINISH* finalise le flux compressé et bloque toute autre tentative de compression. Suite à l'appel de la méthode *flush()* avec l'argument *mode* défini à *Z_FINISH*, la méthode *compress()* ne peut plus être appelée. Il ne reste plus qu'à supprimer l'objet.

`Compress.copy()`

Renvoie une copie de l'objet « compresseur ». Utile pour compresser efficacement un ensemble de données qui partagent un préfixe initial commun.

Les objets décompresseurs prennent en charge les méthodes et attributs suivants :

`Decompress.unused_data`

Un objet *bytes* contenant tous les octets restants après les données compressées. Il vaut donc `b""` tant que des données compressées sont disponibles. Si toute la chaîne d'octets ne contient que des données compressées, il vaut toujours `b""`, un objet *bytes* vide.

`Decompress.unconsumed_tail`

Un objet *bytes* contenant toutes les données non-traitées par le dernier appel à la méthode `decompress()`, à cause d'un dépassement de la limite du tampon de données décompressées. Ces données n'ont pas encore été traitées par la bibliothèque *zlib*, vous devez donc les envoyer (potentiellement en y concaténant encore des données) par un appel à la méthode `decompress()` pour obtenir une sortie correcte.

`Decompress.eof`

Booléen qui signale si la fin du flux compressé est atteint.

Ceci rend possible la distinction entre un flux correctement compressé et un flux incomplet.

Nouveau dans la version 3.3.

`Decompress.decompress(data, max_length=0)`

Décompress *data*, renvoie un objet *bytes*, contenant au moins une partie des données décompressées. Ce résultat doit être concaténé aux résultats des appels précédents à `decompress()`. Des données d'entrée peuvent être conservées dans les tampons internes pour un traitement ultérieur.

Si le paramètre optionnel *max_length* est différent de zéro alors la valeur renvoyée n'est pas plus grande que *max_length*. Cela peut amener à une décompression partielle. Les données non-encore décompressées sont stockées dans l'attribut `unconsumed_tail`. Cette chaîne d'octets doit être transmise à un appel ultérieur à `decompress()`. Si *max_length* vaut zéro, la totalité de l'entrée est décompressée, et l'attribut `unconsumed_tail` reste vide.

Modifié dans la version 3.6 : *max_length* peut être utilisé comme un argument nommé.

`Decompress.flush([length])`

Toutes les entrées en attente sont traitées, et un objet *bytes* est renvoyé, contenant le reste des données à décompresser. Après l'appel à `flush()`, la méthode `decompress()` ne peut pas être rappelée. Il ne reste qu'à détruire l'objet.

Le paramètre optionnel *length* définit la taille initiale du tampon de sortie.

`Decompress.copy()`

Renvoie une copie du décompresseur. Vous pouvez l'utiliser pour sauvegarder l'état de la décompression en cours, afin de pouvoir revenir rapidement à cet endroit plus tard.

Des informations relatives à la version de la bibliothèque *zlib* utilisée sont disponibles via les constantes suivantes :

`zlib.ZLIB_VERSION`

Version de la bibliothèque *zlib* utilisée lors de la compilation du module. Elle peut être différente de la bibliothèque *zlib* actuellement utilisée par le système, qui est consultable par `ZLIB_RUNTIME_VERSION`.

`zlib.ZLIB_RUNTIME_VERSION`

Chaîne contenant la version de la bibliothèque *zlib* actuellement utilisée par l'interpréteur.

Nouveau dans la version 3.3.

Voir aussi :

Module [`gzip`](#) Lire et écrire des fichiers au format **gzip**.

<http://www.zlib.net> Page officielle de la bibliothèque *zlib*.

<http://www.zlib.net/manual.html> La documentation de *zlib* explique le sens et l'utilisation des nombreuses fonctions fournies par la bibliothèque.

13.2 gzip — Support pour les fichiers gzip

Code source : [Lib/gzip.py](#)

Ce module fournit une interface simple pour compresser et décompresser des fichiers tout comme le font les programmes GNU **gzip** et **gunzip**.

La compression de données est fournie par le module *zlib*.

Le module *gzip* fournit la classe *GzipFile* ainsi que les fonctions pratiques *open()*, *compress()* et *decompress()*. La classe *GzipFile* lit et écrit des fichiers au format **gzip**, compressant et décompressant automatiquement les données pour qu'elles aient l'apparence d'un objet *file object* ordinaire.

Notez que les formats de fichier supplémentaires qui peuvent être décompressés par les programmes **gzip** et **gunzip**, comme ceux produits par les programmes **compress** et **pack**, ne sont pas gérés par ce module.

Le module définit les éléments suivants :

`gzip.open(filename, mode='rb', compresslevel=9, encoding=None, errors=None, newline=None)`

Ouvre un fichier compressé en *gzip* en mode binaire ou texte, renvoie un objet *file object*.

L'argument *filename* peut être un nom de fichier (un objet *str* ou *bytes*) ou un objet fichier existant que l'on peut lire, ou où l'on peut écrire.

L'argument *mode* peut-être 'r', 'rb', 'a', 'ab', 'w', 'wb', 'x' ou 'xb' pour le mode binaire ou 'rt', 'at', 'wt', ou 'xt' pour le mode texte. Le mode par défaut est 'rb'.

L'argument *compresslevel* est un entier de 0 à 9, comme pour le constructeur de la classe *GzipFile*.

En mode binaire, cette fonction est équivalente au constructeur de la classe *GzipFile* : `GzipFile(filename, mode, compresslevel)`. Dans ce cas, les arguments *encoding*, *errors* et *newline* ne doivent pas être fournis.

En mode texte, un objet *GzipFile* est créé et empaqueté dans une instance de *io.TextIOWrapper* avec l'encodage, la gestion d'erreur et les fins de ligne spécifiés.

Modifié dans la version 3.3 : Ajout de la prise en charge de *filename* en tant qu'objet *file*, du mode texte et des arguments *encoding*, *errors* et *newline*.

Modifié dans la version 3.4 : Ajout de la prise en charge des modes 'x', 'xb' et 'xt'.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`class gzip.GzipFile(filename=None, mode=None, compresslevel=9, fileobj=None, mtime=None)`

Constructeur de la classe *GzipFile* qui simule la plupart des méthodes d'un objet *file object* à l'exception de la méthode `truncate()`. Au moins un des arguments *fileobj* et *filename* doit avoir une valeur non triviale.

La nouvelle instance de classe est basée sur *fileobj* qui peut être un fichier usuel, un objet *io.BytesIO* ou tout autre objet qui simule un fichier. *fileobj* est par défaut à *None*, dans ce cas *filename* est ouvert afin de fournir un objet fichier.

Quand *fileobj* n'est pas à *None*, l'argument *filename* est uniquement utilisé pour être inclus dans l'entête du fichier **gzip**, qui peut inclure le nom original du fichier décompressé. Il est par défaut défini avec le nom de fichier de *fileobj* s'il est discernable, sinon il est par défaut défini à une chaîne de caractères vide et dans ce cas le nom du fichier original n'est pas inclus dans l'entête.

L'argument *mode* peut-être 'r', 'rb', 'a', 'ab', 'w', 'wb', 'x', ou 'xb', selon que le fichier va être lu ou écrit. Par défaut prend la valeur du mode de *fileobj* si discernable; sinon, la valeur par défaut est 'rb'.

Notez que le fichier est toujours ouvert en mode binaire. Pour ouvrir un fichier compressé en mode texte, utilisez la fonction *open()* (ou empaquetez votre classe *GzipFile* avec un *io.TextIOWrapper*).

L'argument *compresslevel* est un entier de 0 à 9 contrôlant le niveau de compression, "1" est le plus rapide et produit la compression la plus faible et 9 est le plus rapide et produit la compression la plus élevée. 0 désactive la compression. Par défaut à 9.

L'argument *mtime* est un *timestamp* numérique optionnel à écrire dans le champ de date de dernière modification du flux durant la compression. Il ne doit être défini qu'en mode compression. S'il est omis ou `None`, la date courante est utilisée. Voir l'attribut *mtime* pour plus de détails.

Appeler la méthode `close()` d'un objet *GzipFile* ne ferme pas *fileobj*, puisque vous pourriez avoir besoin d'ajouter des informations après les données compressées. Cela vous permet aussi de passer un objet *io.BytesIO* ouvert en écriture en tant que *fileobj* et récupérer le tampon mémoire final en utilisant la méthode `getvalue()` de l'objet *io.BytesIO*.

:La classe *GzipFile* implémente l'interface *io.BufferedIOBase*, incluant l'itération, la déclaration `with`. La méthode `truncate()` est la seule non implémentée.

La classe *GzipFile* fournit aussi la méthode et l'attribut suivant :

peek (*n*)

Lit *n* octets non compressés sans avancer la position dans le fichier. Au plus une seule lecture sur le flux compressé est faite pour satisfaire l'appel. Le nombre d'octets retournés peut être supérieur ou inférieur au nombre demandé.

Note : Bien que l'appel à `peek()` ne change pas la position dans le fichier de l'objet *GzipFile*, il peut changer la position de l'objet de fichier sous-jacent (par exemple, si l'instance de *GzipFile* a été construite avec le paramètre *fileobj*).

Nouveau dans la version 3.2.

mtime

Lors de la décompression, la valeur du champ de date de dernière modification dans le dernier en-tête lu peut être lue à partir de cet attribut, comme un entier. La valeur initiale avant lecture d'un en-tête est `None`.

Tous les flux compressés en **gzip** doivent contenir ce champ *timestamp*. Certains programmes, comme **gunzip**, utilisent ce *timestamp*. Ce format est le même que la valeur retour de `time.time()` et l'attribut `st_mtime` de l'objet retourné par `os.stat()`.

Modifié dans la version 3.1 : Ajout de la prise en charge du mot-clef `with`, ainsi que de l'argument *mtime* du constructeur et de l'attribut *mtime*.

Modifié dans la version 3.2 : Ajout de la prise en charge des fichiers non navigables ou commençant par des octets nuls.

Modifié dans la version 3.3 : La méthode `io.BufferedIOBase.read1()` est désormais implémentée.

Modifié dans la version 3.4 : Ajout de la prise en charge des modes `'x'` et `'xb'`.

Modifié dans la version 3.5 : Ajout de la prise en charge de l'écriture d'objets *bytes-like objects* arbitraires. La méthode `read()` accepte désormais un argument de valeur `None`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

gzip.compress (*data*, *compresslevel*=9)

Comprime les données *data*, renvoie un objet *bytes* contenant les données compressées. L'argument *compresslevel* a la même signification que dans le constructeur de la classe *GzipFile* ci-dessus.

Nouveau dans la version 3.2.

gzip.decompress (*data*)

Décompresse les données *data*, retourne un objet *bytes* contenant les données décompressées.

Nouveau dans la version 3.2.

13.2.1 Exemples d'utilisation

Exemple montrant comment lire un fichier compressé :

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Exemple montrant comment créer un fichier GZIP :

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

Exemple montrant comment compresser dans un GZIP un fichier existant :

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

Exemple montrant comment compresser dans un GZIP un binaire dans une chaîne :

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

Voir aussi :

Module `zlib` Le module de compression de données de base nécessaire pour gérer le format de fichier `gzip`.

13.3 bz2 — Prise en charge de la compression bzip2

Code Source : `Lib/bz2.py`

Ce module fournit une interface complète pour compresser et décompresser les données en utilisant l'algorithme de compression *bzip2*.

Le module `bz2` contient :

- La fonction `open()` et la classe `BZ2File` pour lire et écrire des fichiers compressés.
- Les classes `BZ2Compressor` et `BZ2Decompressor` pour la (dé)compression incrémentielle.
- Les fonctions `compress()` et `decompress()` pour la (dé)compression en une seule fois.

Toutes les classes de ce module peuvent en toute sécurité être accédées depuis de multiples fils d'exécution.

13.3.1 (Dé)compression de fichiers

bz2.open (*filename*, *mode*='r', *compresslevel*=9, *encoding*=None, *errors*=None, *newline*=None)

Ouvre un fichier compressé par *bzip2* en mode binaire ou texte, le renvoyant en *file object*.

Tout comme avec le constructeur pour la classe *BZ2File*, l'argument *filename* peut être un nom de fichier réel (un objet *str* ou *bytes*), ou un objet fichier existant à lire ou à écrire.

L'argument *mode* peut valoir 'r', 'rb', 'w', 'wb', 'x', 'xb', 'a' ou 'ab' pour le mode binaire, ou 'rt', 'wt', 'xt' ou 'at' pour le mode texte. Il vaut par défaut 'rb'.

L'argument *compresslevel* est un entier de 1 à 9, comme pour le constructeur *BZ2File*.

Pour le mode binaire, cette fonction est équivalente au constructeur *BZ2File*: *BZ2File(filename, mode, compresslevel=compresslevel)*. Dans ce cas, les arguments *encoding*, *errors* et *newline* arguments ne doivent pas être fournis.

Pour le mode texte, un objet *BZ2File* est créé et encapsulé dans une instance *io.TextIOWrapper* avec l'encodage spécifié, le comportement de gestion des erreurs et les fins de ligne.

Nouveau dans la version 3.3.

Modifié dans la version 3.4 : Le mode 'x' (création exclusive) est ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object*.

class *bz2.BZ2File* (*filename*, *mode*='r', *buffering*=None, *compresslevel*=9)

Ouvre un fichier *bzip2* en mode binaire.

Si *filename* est un objet *str* ou *bytes*, ouvre le nom de fichier directement. Autrement, *filename* doit être un *file object*, qui est utilisé pour lire ou écrire les données compressées.

L'argument *mode* peut être soit 'r' pour lire (par défaut), 'w' pour écraser, 'x' pour créer exclusivement, ou 'a' pour ajouter. Ils peuvent également être écrits respectivement comme 'rb', 'wb', 'xb' et 'ab'.

Si *filename* est un objet fichier (plutôt que le nom de fichier réel), le mode 'w' ne tronque pas le fichier, mais équivaut à 'a'.

The *buffering* argument is ignored. Its use is deprecated.

If *mode* is 'w' or 'a', *compresslevel* can be a number between 1 and 9 specifying the level of compression : 1 produces the least compression, and 9 (default) produces the most compression.

Si *mode* est 'r', le fichier d'entrée peut être la concaténation de plusieurs flux compressés.

BZ2File fournit tous les membres spécifiés par la classe *io.BufferedIOBase*, excepté les méthodes *detach()* et *truncate()*. L'itération et l'instruction *with* sont prises en charge.

BZ2File fournit aussi la méthode suivante :

peek (*[n]*)

Renvoie des données en mémoire tampon sans avancer la position du fichier. Au moins un octet de donnée (sauf l'EOF) est renvoyé. Le nombre exact d'octets renvoyés n'est pas spécifié.

Note : Bien que l'appel à la méthode *peek()* ne change pas la position du fichier de la classe *BZ2File*, il peut changer la position de l'objet fichier sous-jacent (e.g. si la classe *BZ2File* a été construite en passant un objet fichier à *filename*).

Nouveau dans la version 3.3.

Modifié dans la version 3.1 : La prise en charge de l'instruction *with* a été ajoutée.

Modifié dans la version 3.3 : Les méthodes *fileno()*, *readable()*, *seekable()*, *writable()*, *read1()* et *readinto()* ont été ajoutées.

Modifié dans la version 3.3 : La gestion de *filename* comme *file object* au lieu d'un nom de fichier réel a été ajoutée.

Modifié dans la version 3.3 : Le mode 'a' (ajout) a été ajouté, avec la prise en charge de la lecture des fichiers *multiflux*.

Modifié dans la version 3.4 : Le mode 'x' (création exclusive) est ajouté.

Modifié dans la version 3.5 : La méthode *read()* accepte maintenant un argument *None*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

13.3.2 (Dé)compression incrémentielle

class `bz2.BZ2Compressor` (*compresslevel=9*)

Crée un nouvel objet compresseur. Cet objet peut être utilisé pour compresser les données de manière incrémentielle. Pour une compression en une seule fois, utilisez à la place la fonction `compress()`.

compresslevel, if given, must be a number between 1 and 9. The default is 9.

compress (*data*)

Fournit la donnée à l'objet compresseur. Renvoie un bloc de données compressées si possible, ou autrement une chaîne d'octet vide.

Quand vous avez fini de fournir des données au compresseur, appelez la méthode `flush()` pour finir le processus de compression.

flush ()

Finir le processus de compression. Renvoie la donnée compressée restante dans les tampons internes.

L'objet compresseur ne peut pas être utilisé après que cette méthode a été appelée.

class `bz2.BZ2Decompressor`

Crée un nouvel objet décompresseur. Cet objet peut être utilisé pour décompresser les données de manière incrémentielle. Pour une compression en une seule fois, utilisez à la place la fonction `decompress()`.

Note : Cette classe ne gère pas de manière transparente les entrées contenant plusieurs flux compressés, à la différence de `decompress()` et `BZ2File`. Si vous avez besoin de décompresser une entrée *multiflux* avec la classe `BZ2Decompressor`, vous devez utiliser un nouveau décompresseur pour chaque flux.

decompress (*data*, *max_length=-1*)

Décomprime *data* (un *bytes-like object*), renvoyant une donnée non compressée en tant que chaîne d'octets. Certaines de ces *data* peuvent être mises en interne en tampon, pour un usage lors d'appels ultérieurs par la méthode `decompress()`. La donnée renvoyée doit être concaténée avec la sortie des appels précédents à la méthode `decompress()`.

Si *max_length* est positif, renvoie au plus *max_length* octets de données compressées. Si la limite est atteinte et que d'autres sorties peuvent être produites, l'attribut `needs_input` est positionné sur `False`. Dans ce cas, lors de l'appel suivant à la méthode `decompress()`, vous pouvez fournir `b''` dans *data* afin d'obtenir la suite de la sortie.

Si toutes les données entrées ont été décompressées et renvoyées (soit parce qu'il y avait moins de *max_length* octets, ou parce que *max_length* était négatif), l'attribut `needs_input` sera configuré sur `True`.

Essayer de décompresser des données après que la fin du flux soit atteinte lève une erreur `EOFError`. Toute donnée trouvée après la fin du flux est ignorée et sauvegardée dans l'attribut `unused_data`.

Modifié dans la version 3.5 : Ajout du paramètre *max_length*.

eof

`True` si le marqueur de fin de flux a été atteint.

Nouveau dans la version 3.3.

unused_data

Donnée trouvée après la fin du flux compressé.

Si l'attribut est accédé avant que la fin du flux ait été atteint, sa valeur sera `b''`.

needs_input

`False` si la méthode `decompress()` peut fournir plus de données décompressées avant l'acquisition d'une nouvelle entrée non compressée.

Nouveau dans la version 3.5.

13.3.3 (Dé)compression en une fois

`bz2.compress(data, compresslevel=9)`

Comprime *data*.

compresslevel, if given, must be a number between 1 and 9. The default is 9.

Pour la compression incrémentielle, utilisez à la place la classe *BZ2Compressor*.

`bz2.decompress(data)`

Décompresse *data*.

Si *data* est la concaténation de multiples flux compressés, décompresse tous les flux.

Pour une décompression incrémentielle, utilisez à la place la classe *BZ2Decompressor*.

Modifié dans la version 3.3 : Prise en charge des entrées *multiflux*.

13.4 lzma — Compression via l'algorithme LZMA

Nouveau dans la version 3.3.

Code source : [Lib/lzma.py](#)

Ce module fournit des classes et des fonctions utiles pour compresser et décompresser des données en utilisant l'algorithme de compression LZMA. Ce module inclut aussi une interface prenant en charge les fichiers *.xz* et son format originel *.lzma* utilisés par l'utilitaire *xz*, ainsi que les flux bruts compressés.

L'interface disponible par ce module ressemble en de nombreux points à celle du module *bz2*. Cependant, notez que la *LZMAFile* n'est pas *thread-safe*, comme l'est la *bz2.BZ2File*. Donc, si vous souhaitez utiliser une seule instance de *LZMAFile* pour plusieurs fils, il sera alors nécessaire de la protéger avec un verrou (*lock*).

exception `lzma.LZMAError`

Cette exception est levée dès lors qu'une erreur survient pendant la compression ou la décompression, ou pendant l'initialisation de l'état de la compression/décompression.

13.4.1 Lire et écrire des fichiers compressés

`lzma.open(filename, mode="rb", *, format=None, check=-1, preset=None, filters=None, encoding=None, errors=None, newline=None)`

Ouvre un fichier compressé sous LZMA en mode binaire ou texte, retourne un *file object*.

L'argument *nom de fichier* peut être soit le nom d'un fichier à créer (donné pour *str*, *bytes* ou un objet *path-like*), dont le fichier nommé reste ouvert, ou soit un objet fichier existant à lire ou à écrire.

L'argument *mode* peut être n'importe quel argument suivant : "r", "rb", "w", "wb", "x", "xb", "a" ou "ab" pour le mode binaire, ou "rt", "wt", "xt", ou "at" pour le mode texte. La valeur par défaut est "rb".

Quand un fichier est ouvert pour le lire, les arguments *format* et *filters* ont les mêmes significations que pour la *LZMAdecompressor*. Par conséquent, les arguments *check* et *preset* ne devront pas être sollicités.

Dès ouverture d'un fichier pour l'écriture, les arguments *format*, *check*, *preset* et *filters* ont le même sens que dans la *LZMACompressor*.

Pour le mode binaire, cette fonction équivaut à la *LZMAFile* constructor : `LZMAFile(filename, mode, ...)`. Dans ce cas précis, les arguments *encoding*, *errors* et *newline* ne sont pas accessibles.

Pour le mode texte, un objet de *LZMAFile* est créé, et enveloppé dans une instance de *io.TextIOWrapper* avec son encodage spécifique, son gestionnaire d'erreur, et ses signes pour les renvois à la ligne.

Modifié dans la version 3.4 : Support ajouté pour les modes "x", "xb" and "xt".

Modifié dans la version 3.6 : Accepte un *path-like object*.

class `lzma.LZMAFile` (*filename=None, mode="r", *, format=None, check=-1, preset=None, filters=None*)

Ouvre un fichier LZMA compressé en mode binaire.

An *LZMAFile* can wrap an already-open *file object*, or operate directly on a named file. The *filename* argument specifies either the file object to wrap, or the name of the file to open (as a *str*, *bytes* or *path-like object*). When wrapping an existing file object, the wrapped file will not be closed when the *LZMAFile* is closed.

L'argument *mode* peut être soit "r" pour la lecture (défaut), "w" pour la ré-écriture, "x" pour la création exclusive, ou "a" pour l'insertion. Elles peuvent aussi être écrites de la façon suivante : "rb", "wb", "xb" et "ab" respectivement.

Si l'argument *filename* est un objet fichier (plutôt qu'un nom de fichier courant), un mode "w" ne tronquera aucunement le fichier, et sera équivalent à "a".

Dès l'ouverture d'un fichier pour être lu, le fichier d'entrée peut être le résultat d'une concaténation de plusieurs flux distincts et compressés. Ceux-ci sont décodés de manière transparente en un seul flux logique.

Quand un fichier est ouvert pour le lire, les arguments *format* et *filters* ont les mêmes significations que pour la *LZMADecompressor*. Par conséquent, les arguments *check* et *preset* ne devront pas être sollicités.

Dès ouverture d'un fichier pour l'écriture, les arguments *format*, *check*, *preset* et *filters* ont le même sens que dans la *LZMACompressor*.

La *LZMAFile* supporte tous les attributs précisés dans *io.BufferedIOBase*, sauf *detach()* et *truncate()*. Le *with* et son itération sont également supportés.

Les méthodes suivantes sont aussi disponibles :

peek (*size=-1*)

Renvoie la donnée en mémoire-tampon sans progression de la position du fichier. Au moins un octet de donnée sera renvoyé, jusqu'à ce que l'EOF soit atteinte. Le nombre exact d'octets renvoyés demeure indéterminé (l'argument *taille* est ignoré).

Note : Alors que l'exécution de *peek()* ne change pas la position du fichier de la classe *LZMAFile*, la position de l'objet sous-jacent peut être modifiée (ex. : si la classe *LZMAFile* est construite en lui transmettant un fichier-objet comme *filename*).

Modifié dans la version 3.4 : Support ajouté pour les modes "x" and "xb".

Modifié dans la version 3.5 : La méthode *read()* accepte maintenant un argument *None*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

13.4.2 Compresser et décompresser une donnée en mémoire

class `lzma.LZMACompressor` (*format=FORMAT_XZ, check=-1, preset=None, filters=None*)

Crée un objet compresseur, qui peut être utilisé pour compresser incrémentalement une donnée.

Pour une façon plus adaptée de compresser un seul extrait de donnée, voir *compress()*.

L'argument *format* définit quel format de conteneur sera mis en œuvre. Les valeurs possibles sont :

- **FORMAT_XZ** : Le format du conteneur **.xz**. C'est le format par défaut.
- **FORMAT_ALONE** : L'ancien format du conteneur **.lzma**. Ce format est davantage limité que **.xz** –il ne supporte pas les vérifications d'intégrité ou les filtres multiples.
- **FORMAT_RAW** : Un flux de données brut, n'utilisant aucun format de conteneur. Ce format spécifique ne prend pas en charge les vérifications d'intégrité et exige systématiquement la définition d'une chaîne de filtrage personnalisée (à la fois pour la compression et la décompression). Par ailleurs, les données compressées par ce biais ne peuvent pas être décompressées par l'usage de **FORMAT_AUTO** (voir : *LZMADecompressor*).

L'argument *check* détermine le type de vérification d'intégrité à exploiter avec la donnée compressée. Cette vérification est déclenchée lors de la décompression, pour garantir que la donnée n'a pas été corrompue. Les valeurs possibles sont :

- **CHECK_NONE** : Pas de vérification d'intégrité. C'est la valeur par défaut (et la seule valeur acceptable) pour **FORMAT_ALONE** et **FORMAT_RAW**.

- `CHECK_CRC32` : Vérification par Redondance Cyclique 32-bit (*Cyclic Redundancy Check*).
- `CHECK_CRC64` : Vérification par Redondance Cyclique 64-bit (*Cyclic Redundancy Check*). Valeur par défaut pour `FORMAT_XZ`.
- `CHECK_SHA256` : Algorithme de Hachage Sécurisé 256-bit (*Secure Hash Algorithm*).

Si le type de vérification n'est pas supporté par le système, une erreur de type `LZMAError` est levée.

Les réglages de compression peuvent être définis soit comme un pré-réglage de niveau de compression (avec l'argument *preset*); soit de façon détaillée comme une chaîne particulière de filtres (avec l'argument *filters*).

L'argument *preset* (s'il est fourni) doit être un entier compris entre 0 et 9 (inclus), éventuellement relié à OR avec la constante `PRESET_EXTREME`. Si aucun *preset* ni *filters* ne sont définis, le comportement par défaut consiste à utiliser la `PRESET_DEFAULT` (niveau par défaut : 6). Des pré-réglages plus élevés entraînent une sortie plus petite, mais rend le processus de compression plus lent.

Note : En plus d'être plus gourmande en CPU, la compression avec des pré-réglages plus élevés nécessite beaucoup plus de mémoire (et produit des résultats qui nécessitent plus de mémoire pour décompresser). Par exemple, avec le pré-réglage 9, l'objet d'une `LZMACompressor` peut dépasser largement les 800 Mo. Pour cette raison, il est généralement préférable de respecter le pré-réglage par défaut.

L'argument *filters* (s'il est défini) doit être un critère de la chaîne de filtrage. Voir *Préciser des chaînes de filtre personnalisées* pour plus de précisions.

compress (*data*)

Une *data* compressée (un objet *bytes*), renvoie un objet *bytes* contenant une donnée compressée pour au moins une partie de l'entrée. Certaines *data* peuvent être mise en tampon, pour être utilisées lors de prochains appels par `compress()` et `flush()`. La donnée renvoyée pourra être concaténée avec la sortie d'appels précédents vers la méthode `compress()`.

flush ()

Conclut l'opération de compression, en renvoyant l'objet *bytes* constitué de toutes les données stockées dans les tampons interne du compresseur.

Le compresseur ne pourra pas être sollicité après l'appel de cette méthode.

class `lzma.LZMADecompressor` (*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

Crée un objet de décompression, pour décompresser de façon incrémentale une donnée.

Pour un moyen plus pratique de décompresser un flux compressé complet en une seule fois, voir `decompress()`.

L'argument *format* spécifie le format du conteneur à utiliser. La valeur par défaut est `FORMAT_AUTO` pouvant à la fois décompresser les fichiers `.xz` and `.lzma`. D'autres valeurs sont possibles comme `FORMAT_XZ`, `FORMAT_ALONE`, et `FORMAT_RAW`.

L'argument *memlimit* spécifie une limite (en octets) sur la quantité de mémoire que le décompresseur peut utiliser. Lorsque cet argument est utilisé, la décompression échouera avec une `LZMAError` s'il n'est pas possible de décompresser l'entrée dans la limite mémoire disponible.

L'argument *filters* spécifie la chaîne de filtrage utilisée pour créer le flux décompressé. Cet argument est requis si *format* est `FORMAT_RAW`, mais ne doit pas être utilisé pour d'autres formats. Voir *Préciser des chaînes de filtre personnalisées* pour plus d'informations sur les chaînes de filtrage.

Note : Cette classe ne gère pas de manière transparente les entrées contenant plusieurs flux compressés, contrairement à `decompress()` et `LZMAFile`. Pour décompresser une entrée multi-flux avec `LZMADecompressor`, vous devez créer un nouveau décompresseur à chaque flux.

decompress (*data*, *max_length=-1*)

Décompresse *data* (un *bytes-like object*), renvoyant une donnée non compressée en tant que chaîne d'octets. Certaines de ces *data* peuvent être mises en tampon interne, pour un usage lors d'appels ultérieurs par la méthode `decompress()`. La donnée renvoyée doit être concaténée avec la sortie des appels précédents à la méthode `decompress()`.

Si *max_length* est positif, renvoie au plus *max_length* octets de données compressées. Si la limite est atteinte et que d'autres sorties peuvent être produites, l'attribut `needs_input` est positionné sur `False`. Dans ce

cas, lors de l'appel suivant à la méthode `decompress()`, vous pouvez fournir `b''` dans `data` afin d'obtenir la suite de la sortie.

Si toutes les données entrées ont été décompressées et renvoyées (soit parce qu'il y avait moins de `max_length` octets, ou parce que `max_length` était négatif), l'attribut `needs_input` sera configuré sur `True`.

Essayer de décompresser des données après que la fin du flux soit atteinte lève une erreur `EOFError`. Toute donnée trouvée après la fin du flux est ignorée et sauvegardée dans l'attribut `unused_data`.

Modifié dans la version 3.5 : Ajout du paramètre `max_length`.

check

L'ID de la vérification d'intégrité exploité par le flux entrant. Il s'agit de `CHECK_UNKNOWN` tant que ce flux a été décodé pour déterminer quel type de vérification d'intégrité à été utilisé.

eof

`True` si le marqueur de fin de flux a été atteint.

unused_data

Donnée trouvée après la fin du flux compressé.

Avant d'atteindre la fin du flux, ce sera `b''`.

needs_input

`False` si la méthode `decompress()` peut fournir plus de données décompressées avant l'acquisition d'une nouvelle entrée non compressée.

Nouveau dans la version 3.5.

`lzma.compress(data, format=FORMAT_XZ, check=-1, preset=None, filters=None)`

`data` compressée (un objet `bytes`), renvoyant une donnée compressée comme un objet `bytes`.

Voir `LZMACompressor` ci-dessus pour une description des arguments `format`, `check`, `preset` and `filters`.

`lzma.decompress(data, format=FORMAT_AUTO, memlimit=None, filters=None)`

Décompresse `data` (un objet `bytes`), et retourne la donnée décompressée sous la forme d'un objet `bytes`.

Si `data` est le résultat de la concaténation de plusieurs flux compressés et distincts, il les décompresse tous, et retourne les résultats concaténés.

Voir `LZMADecompressor` ci-dessus pour une description des arguments `format`, `memlimit` et `filters`.

13.4.3 Divers

`lzma.is_check_supported(check)`

Renvoie vrai si le contrôle d'intégrité donné est supporté par le système.

`CHECK_NONE` et `CHECK_CRC32` sont toujours supportés. `CHECK_CRC64` et `CHECK_SHA256` peuvent être indisponibles si vous utilisez une version de `liblzma` compilée avec des possibilités restreintes.

13.4.4 Préciser des chaînes de filtre personnalisées

Une chaîne de filtres est une séquence de dictionnaires, où chaque dictionnaire contient l'ID et les options pour chaque filtre. Le moindre dictionnaire contient la clé `"id"` et peut aussi contenir d'autres clés pour préciser chaque options relative au filtre déclaré. Les ID valides des filtres sont définies comme suit :

— Filtres de compression :

— `FILTER_LZMA1` (à utiliser avec `FORMAT_ALONE`)

— `FILTER_LZMA2` (à utiliser avec `FORMAT_XZ` et `FORMAT_RAW`)

— Filtre Delta :

— `FILTER_DELTA`

— Filtres Branch-Call-Jump (BCJ) :

— `FILTER_X86`

— `FILTER_IA64`

— `FILTER_ARM`

- FILTER_ARMTHUMB
- FILTER_POWERPC
- FILTER_SPARC

Une chaîne de filtres peut contenir jusqu'à 4 filtres, et ne peut pas être vide. Le dernier filtre de cette chaîne devra être un filtre de compression, et tous les autres doivent être des filtres delta ou BCJ.

Les filtres de compression contiennent les options suivantes (définies comme entrées additionnelles dans le dictionnaire qui représente le filtre) :

- `preset` : Un pré-réglage à exploiter comme une source de valeurs par défaut pour les options qui ne sont pas explicitement définies.
- `dict_size` : La taille du dictionnaire en octets. Comprise entre 4 Ko et 1.5 Go (inclus).
- `lc` : Nombre de bits dans le contexte littéral.
- `lp` : Nombre de bits dans la position littérale. La somme `lc` + `lp` devra être au moins 4.
- `pb` : Nombre de bits à cette position ; au moins 4.
- `mode` : `MODE_FAST` ou `MODE_NORMAL`.
- `nice_len` : Ce qui devra être pris en compte comme « longueur appréciable » pour une recherche. Il devra être 273 ou moins.
- `mf` : Quel type d'index de recherche à utiliser – `MF_HC3`, `MF_HC4`, `MF_BT2`, `MF_BT3`, ou `MF_BT4`.
- `depth` : Profondeur maximum de la recherche, utilisée par l'index de recherche. 0 (défaut) signifie une sélection automatique basée sur des options de filtres différents.

Le filtre delta stocke les différences entre octets, induisant davantage d'entrées répétitives pour le compresseur, selon les circonstances. Il supporte une option, `dist`. Ce paramètre définit la distance entre les octets à soustraire. Par défaut : 1, soit la différence entre des octets adjacents.

Les filtres BCJ sont conçus pour être appliqués sur du langage machine. Ils convertissent les branches relatives, les appels et les sauts dans le code à des fins d'adressage strict, dans le but d'augmenter la redondance mise en jeu par le compresseur. Ils ne supportent qu'une seule option : `start_offset`, pour définir l'adresse où sera déclenché le début de la donnée d'entrée. Par défaut : 0.

13.4.5 Exemples

Lire un fichier compressé :

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Créer un fichier compressé :

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Compresser des données en mémoire :

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

Compression incrémentale :

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
```

(suite sur la page suivante)

(suite de la page précédente)

```

out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])

```

Ecrire des données compressées dans un fichier préalablement ouvert :

```

import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")

```

Créer un fichier compressé en utilisant une chaîne de filtre personnalisée :

```

import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")

```

13.5 zipfile — Travailler avec des archives ZIP

Code source : [Lib/zipfile.py](#)

Le format de fichier ZIP est une archive et un standard de compression couramment utilisés. Ce module fournit des outils pour créer, écrire, ajouter des données à et lister un fichier ZIP. L'utilisation avancée de ce module requiert une certaine compréhension du format, comme défini dans [PKZIP Application Note](#).

Ce module ne gère pas pour l'instant les fichiers ZIP multi-disque. Il gère les fichiers ZIP qui utilisent les extensions ZIP64 (c'est-à-dire des fichiers d'une taille supérieure à 4 Go). Il gère le chiffrement d'archives ZIP chiffrées, mais il ne peut pas pour l'instant créer de fichier chiffré. Le déchiffrement est extrêmement lent car il est implémenté uniquement en Python plutôt qu'en C.

Le module définit les éléments suivants :

exception `zipfile.BadZipFile`

Erreur levée en cas de fichier ZIP non valide.

Nouveau dans la version 3.2.

exception `zipfile.BadZipfile`

Alias de `BadZipFile`, pour la compatibilité avec les versions de Python précédentes.

Obsolète depuis la version 3.2.

exception `zipfile.LargeZipFile`

Erreur levée quand un fichier ZIP nécessite la fonctionnalité ZIP64 mais qu'elle n'a pas été activée.

class `zipfile.ZipFile`

Classe pour lire et écrire des fichiers ZIP. Voir la section [Objets ZipFile](#) pour les détails du constructeur.

class zipfile.PyZipFile

Classe pour créer des archives ZIP contenant des bibliothèques Python.

class zipfile.ZipInfo (filename='NoName', date_time=(1980, 1, 1, 0, 0, 0))

Classe utilisée pour représenter les informations d'un membre d'une archive. Les instances de cette classe sont retournées par les méthodes `getinfo()` et `infolist()` des objets `ZipFile`. La plupart des utilisateurs du module `zipfile` n'ont pas besoin de créer ces instances mais d'utiliser celles créées par ce module. `filename` doit être le nom complet du membre de l'archive et `date_time` doit être un *tuple* contenant six champs qui décrit la date de dernière modification du fichier ; les champs sont décrits dans la section *Objets ZipInfo*.

`zipfile.is_zipfile (filename)`

Retourne `True` si `filename` est un fichier ZIP valide basé sur son nombre magique, sinon retourne `False`. `filename` peut aussi être un fichier ou un objet fichier-compatible.

Modifié dans la version 3.1 : Gestion des objets fichier et fichier-compatibles.

`zipfile.ZIP_STORED`

Constante numérique pour un membre d'une archive décompressée.

`zipfile.ZIP_DEFLATED`

Constante numérique pour la méthode habituelle de compression de ZIP. Nécessite le module `zlib`.

`zipfile.ZIP_BZIP2`

Constante numérique pour la méthode de compressions BZIP2. Nécessite le module `bz2`.

Nouveau dans la version 3.3.

`zipfile.ZIP_LZMA`

Constante numérique pour la méthode de compressions LZMA. Nécessite le module `lzma`.

Nouveau dans la version 3.3.

Note : La spécification du format de fichier ZIP inclut la gestion de la compression BZIP2 depuis 2001 et LZMA depuis 2006. Néanmoins, certains outils (comme certaines versions de Python) ne gèrent pas ces méthodes de compression et peuvent soit totalement refuser de traiter le fichier ZIP soit ne pas extraire certains fichiers.

Voir aussi :

PKZIP Application Note Documentation sur le format de fichier ZIP par Phil Katz, créateur du format et des algorithmes utilisés.

Info-ZIP Home Page Informations sur les programmes et les bibliothèques de développement d'archivage ZIP du projet Info-ZIP.

13.5.1 Objets ZipFile

class zipfile.ZipFile (file, mode='r', compression=ZIP_STORED, allowZip64=True)

Open a ZIP file, where `file` can be a path to a file (a string), a file-like object or a *path-like object*. The `mode` parameter should be `'r'` to read an existing file, `'w'` to truncate and write a new file, `'a'` to append to an existing file, or `'x'` to exclusively create and write a new file. If `mode` is `'x'` and `file` refers to an existing file, a `FileExistsError` will be raised. If `mode` is `'a'` and `file` refers to an existing ZIP file, then additional files are added to it. If `file` does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as `python.exe`). If `mode` is `'a'` and the file does not exist at all, it is created. If `mode` is `'r'` or `'a'`, the file should be seekable. `compression` is the ZIP compression method to use when writing the archive, and should be `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2` or `ZIP_LZMA`; unrecognized values will cause `NotImplementedError` to be raised. If `ZIP_DEFLATED`, `ZIP_BZIP2` or `ZIP_LZMA` is specified but the corresponding module (`zlib`, `bz2` or `lzma`) is not available, `RuntimeError` is raised. The default is `ZIP_STORED`. If `allowZip64` is `True` (the default) zipfile will create ZIP files that use the ZIP64 extensions when

the zipfile is larger than 4 GiB. If it is false *zipfile* will raise an exception when the ZIP file would require ZIP64 extensions.

Si le fichier est créé avec le mode 'w', 'x' ou 'a' et ensuite *fermé* sans ajouter de fichiers à l'archive, la structure appropriée pour un fichier archive ZIP vide sera écrite dans le fichier.

ZipFile is also a context manager and therefore supports the *with* statement. In the example, *myzip* is closed after the *with* statement's suite is finished—even if an exception occurs :

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

Nouveau dans la version 3.2 : Ajout de la possibilité d'utiliser *ZipFile* comme un gestionnaire de contexte.

Modifié dans la version 3.3 : Ajout de la gestion de la compression *bzip2* et *lzma*.

Modifié dans la version 3.4 : Les extensions ZIP64 sont activées par défaut.

Modifié dans la version 3.5 : Ajout de la gestion de l'écriture dans des flux non navigables. Ajout de la gestion du mode x.

Modifié dans la version 3.6 : Auparavant, une simple exception *RuntimeError* était levée pour des valeurs de compression non reconnues.

Modifié dans la version 3.6.2 : Le paramètre *file* accepte un objet fichier-compatible *path-like object*.

ZipFile.close()

Ferme l'archive. Vous devez appeler *close()* avant de terminer votre programme ou des informations essentielles n'y seront pas enregistrées.

ZipFile.getinfo(name)

Retourne un objet *ZipInfo* avec les informations du membre *name* de l'archive. Appeler *getinfo()* pour un nom non contenu dans l'archive lève une exception *KeyError*.

ZipFile.infolist()

Retourne une liste contenant un objet *ZipInfo* pour chaque membre de l'archive. Les objets ont le même ordre que leurs entrées dans le fichier ZIP présent sur disque s'il s'agissait d'une archive préexistante.

ZipFile.namelist()

Retourne une liste des membres de l'archive indexés par leur nom.

ZipFile.open(name, mode='r', pwd=None, *, force_zip64=False)

Accède un membre de l'archive en tant qu'objet fichier-compatible binaire. *name* peut être soit le nom d'un fichier au sein de l'archive soit un objet *ZipInfo*. Le paramètre *mode*, si inclus, doit être défini à 'r' (valeur par défaut) ou 'w'. *pwd* est le mot de passe utilisé pour déchiffrer des fichiers ZIP chiffrés.

open() est aussi un gestionnaire de contexte et gère ainsi la déclaration *with* :

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

With mode 'r' the file-like object (*ZipExtFile*) is read-only and provides the following methods : *read()*, *readline()*, *readlines()*, *__iter__()*, *__next__()*. These objects can operate independently of the *ZipFile*.

Avec mode='w' un descripteur de fichier en écriture est retourné, gérant la méthode *write()*. Quand le descripteur d'un fichier inscriptible est ouvert, tenter de lire ou écrire d'autres fichiers dans le fichier ZIP lève une exception *ValueError*.

Lors de l'écriture d'un fichier, si la taille du fichier n'est pas connue mais peut être supérieure à 2 GiO, spécifiez *force_zip64=True* afin de vous assurer que le format d'en-tête est capable de supporter des fichiers volumineux. Si la taille du fichier est connue à l'avance, instanciez un objet *ZipInfo* avec l'attribut *file_size* défini et utilisez-le en tant que paramètre *name*.

Note : Les méthodes *open()*, *read()* et *extract()* peuvent prendre un nom de fichier ou un objet *ZipInfo*. Cela est appréciable lorsqu'on essaie de lire un fichier ZIP qui contient des membres avec des noms en

double.

Modifié dans la version 3.6 : Suppression de la gestion de `mode='U'`. Utilisez `io.TextIOWrapper` pour lire des fichiers texte compressés en mode *universal newlines*.

Modifié dans la version 3.6 : La méthode `open()` peut désormais être utilisée pour écrire des fichiers dans l'archive avec l'option `mode='w'`.

Modifié dans la version 3.6 : Appeler `open()` sur un fichier `ZipFile` fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

`ZipFile.extract(member, path=None, pwd=None)`

Extrait un membre de l'archive dans le répertoire courant ; *member* doit être son nom complet ou un objet `ZipInfo`. Ses propriétés de fichier sont extraites le plus fidèlement possible. *path* spécifie un répertoire différent où l'extraire. *member* peut être un nom de fichier ou un objet `ZipInfo`. *pwd* est le mot de passe utilisé pour les fichiers chiffrés. Retourne le chemin normalisé créé (un dossier ou un nouveau fichier).

Note : Si le nom de fichier d'un membre est un chemin absolu, le disque/partage UNC et les (anti)slashes de départ seront supprimés, par exemple `///foo/bar` devient `foo/bar` sous Unix et `C:\foo\bar` devient `foo\bar` sous Windows. Et tous les composants `".."` dans le nom de fichier d'un membre seront supprimés, par exemple `../foo../../ba..r` devient `foo../ba..r`. Sous Windows les caractères illégaux (`:`, `<`, `>`, `|`, `"`, `?` et `*`) sont remplacés par un *underscore* (`_`).

Modifié dans la version 3.6 : Appeler `extract()` sur un fichier `ZipFile` fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

Modifié dans la version 3.6.2 : Le paramètre *path* accepte un objet chemin-compatible *path-like object*.

`ZipFile.extractall(path=None, members=None, pwd=None)`

Extrait tous les membres de l'archive dans le répertoire courant. *path* spécifie un dossier de destination différent. *members* est optionnel et doit être un sous-ensemble de la liste retournée par `namelist()`. *pwd* est le mot de passe utilisé pour les fichiers chiffrés.

Avertissement : N'extrayez jamais d'archives depuis des sources non fiables sans inspection préalable. Il est possible que des fichiers soient créés en dehors de *path*, par exemple des membres qui ont des chemins de fichier absolus commençant par `"/"` ou des noms de fichier avec deux points `".."`. Ce module essaie de prévenir ceci. Voir la note de `extract()`.

Modifié dans la version 3.6 : Appeler `extractall()` sur un fichier `ZipFile` fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

Modifié dans la version 3.6.2 : Le paramètre *path* accepte un objet chemin-compatible *path-like object*.

`ZipFile.printdir()`

Affiche la liste des contenus de l'archive sur `sys.stdout`.

`ZipFile.setpassword(pwd)`

Définit *pwd* comme mot de passe par défaut pour extraire des fichiers chiffrés.

`ZipFile.read(name, pwd=None)`

Retourne les octets du fichier *name* dans l'archive. *name* est le nom du fichier dans l'archive ou un objet `ZipInfo`. L'archive doit être ouverte en mode lecture ou ajout de données. *pwd* est le mot de passe utilisé pour les fichiers chiffrés et, si spécifié, écrase le mot de passe par défaut défini avec `setpassword()`. Appeler `read()` sur un fichier `ZipFile` qui utilise une méthode de compression différente de `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2` ou `ZIP_LZMA` lève une erreur `NotImplementedError`. Une erreur est également levée si le module de compression n'est pas disponible.

Modifié dans la version 3.6 : Appeler `read()` sur un fichier `ZipFile` fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

ZipFile.testzip()

Lit tous les fichiers de l'archive et vérifie leurs sommes CRC et leurs en-têtes. Retourne le nom du premier fichier mauvais ou retourne `None` sinon.

Modifié dans la version 3.6 : Calling `testzip()` on a closed `ZipFile` will raise a `ValueError`. Previously, a `RuntimeError` was raised.

ZipFile.write(filename, arcname=None, compress_type=None)

Write the file named `filename` to the archive, giving it the archive name `arcname` (by default, this will be the same as `filename`, but without a drive letter and with leading path separators removed). If given, `compress_type` overrides the value given for the `compression` parameter to the constructor for the new entry. The archive must be open with mode `'w'`, `'x'` or `'a'`.

Note : Les noms d'archive doivent être relatifs à la racine de l'archive, c'est-à-dire qu'ils ne doivent pas commencer par un séparateur de chemin.

Note : Si `arcname` (ou `filename` si `arcname` n'est pas donné) contient un octet nul, le nom du fichier dans l'archive sera tronqué à l'octet nul.

Modifié dans la version 3.6 : Appeler `write()` sur un fichier `ZipFile` fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

ZipFile.writestr(zinfo_or_arcname, data[, compress_type])

Write a file into the archive. The contents is `data`, which may be either a `str` or a `bytes` instance; if it is a `str`, it is encoded as UTF-8 first. `zinfo_or_arcname` is either the file name it will be given in the archive, or a `ZipInfo` instance. If it's an instance, at least the filename, date, and time must be given. If it's a name, the date and time is set to the current date and time. The archive must be opened with mode `'w'`, `'x'` or `'a'`.

If given, `compress_type` overrides the value given for the `compression` parameter to the constructor for the new entry, or in the `zinfo_or_arcname` (if that is a `ZipInfo` instance).

Note : Lorsque l'on passe une instance de `ZipInfo` dans le paramètre `zinfo_or_arcname`, la méthode de compression utilisée sera celle spécifiée dans le membre `compress_type` de l'instance `ZipInfo` donnée. Par défaut, le constructeur de la classe `ZipInfo` définit ce membre à `ZIP_STORED`.

Modifié dans la version 3.2 : L'argument `compress_type`.

Modifié dans la version 3.6 : Appeler `writestr()` sur un fichier `ZipFile` fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

Les attributs suivants sont aussi disponibles :

ZipFile.filename

Nom du fichier ZIP.

ZipFile.debug

Le niveau d'affichage de `debug` à utiliser. Peut être défini de 0 (par défaut, pas d'affichage) à 3 (affichage le plus bavard). Les informations de débogage sont affichées sur `sys.stdout`.

ZipFile.comment

The comment associated with the ZIP file as a `bytes` object. If assigning a comment to a `ZipFile` instance created with mode `'w'`, `'x'` or `'a'`, it should be no longer than 65535 bytes. Comments longer than this will be truncated.

13.5.2 Objets *PyZipFile*

Le constructeur de *PyZipFile* prend les mêmes paramètres que le constructeur de *ZipFile* avec un paramètre additionnel *optimize*.

class `zipfile.PyZipFile` (*file*, *mode*='r', *compression*=ZIP_STORED, *allowZip64*=True, *optimize*=-1)

Nouveau dans la version 3.2 : Le paramètre *optimize*.

Modifié dans la version 3.4 : Les extensions ZIP64 sont activées par défaut.

Les instances ont une méthode supplémentaire par rapport aux objets *ZipFile* :

writepy (*pathname*, *basename*="", *filterfunc*=None)

Cherche les fichiers *.py et ajoute le fichier correspondant à l'archive.

Si le paramètre *optimize* du constructeur de *PyZipFile* n'a pas été donné ou est à -1, le fichier correspondant est un fichier *.pyc, à compiler si nécessaire.

Si le paramètre *optimize* du constructeur de *PyZipFile* est à 0, 1 ou 2, ne sont ajoutés dans l'archive que les fichiers avec ce niveau d'optimisation (voir *compile()*), à compiler si nécessaire.

If *pathname* is a file, the filename must end with .py, and just the (corresponding *.pyc) file is added at the top level (no path information). If *pathname* is a file that does not end with .py, a *RuntimeError* will be raised. If it is a directory, and the directory is not a package directory, then all the files *.pyc are added at the top level. If the directory is a package directory, then all *.pyc are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively.

basename n'est sensé être utilisé qu'en interne.

filterfunc, si donné, doit être une fonction prenant une seule chaîne de caractères en argument. Il lui sera passé chaque chemin (incluant chaque chemin de fichier complet individuel) avant d'être ajouté à l'archive. Si *filterfunc* retourne une valeur fausse, le chemin n'est pas ajouté et si c'est un répertoire son contenu est ignoré. Par exemple, si nos fichiers de test sont tous soit dans des répertoires *test* ou commencent par *test_*, nous pouvons utiliser une fonction *filterfunc* pour les exclure :

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
>>> zf.writepy('myprog', filterfunc=notests)
```

La méthode *writepy()* crée des archives avec des noms de fichier comme suit :

```
string.pyc                # Top level name
test/__init__.pyc         # Package directory
test/testall.pyc          # Module test.testall
test/bogus/__init__.pyc   # Subpackage directory
test/bogus/myfile.pyc     # Submodule test.bogus.myfile
```

Nouveau dans la version 3.4 : Le paramètre *filterfunc*.

Modifié dans la version 3.6.2 : Le paramètre *pathname* accepte un objet chemin-compatible *path-like object*.

13.5.3 Objets *ZipInfo*

Des instances de la classe *ZipInfo* sont retournées par les méthodes *getinfo()* et *infolist()* des objets *ZipFile*. Chaque objet stocke des informations sur un seul membre de l'archive ZIP.

Il y a une méthode de classe pour créer une instance de *ZipInfo* pour un fichier du système de fichiers :

classmethod `ZipInfo.from_file` (*filename*, *arcname*=None)

Construit une instance de *ZipInfo* pour le fichier du système de fichiers, en préparation de l'ajouter à un fichier ZIP.

filename doit être un chemin vers un fichier ou un répertoire dans le système de fichiers.

Si *arcname* est spécifié, il est utilisé en tant que nom dans l'archive. Si *arcname* n'est pas spécifié, le nom sera le même que *filename* mais sans lettre de disque et sans séparateur de chemin en première position.

Nouveau dans la version 3.6.

Modifié dans la version 3.6.2 : Le paramètre *filename* accepte un objet chemin-compatible *path-like object*.

Les instances ont les méthodes et attributs suivants :

`ZipInfo.is_dir()`

Retourne `True` si le membre d'archive est un répertoire.

Utilise le nom de l'entrée : les répertoires doivent toujours se terminer par `/`.

Nouveau dans la version 3.6.

`ZipInfo.filename`

Nom du fichier dans l'archive.

`ZipInfo.date_time`

Date et heure de dernière modification pour le membre de l'archive. *Tuple* de six valeurs :

Index	Valeur
0	Année (≥ 1980)
1	Mois (indexé à partir de 1)
2	Jour du mois (indexé à partir de 1)
3	Heures (indexées à partir de 0)
4	Minutes (indexées à partir de 0)
5	Secondes (indexées à partir de 0)

Note : Le format de fichier ZIP ne gère pas les horodatages avant 1980.

`ZipInfo.compress_type`

Type de compression du membre d'archive.

`ZipInfo.comment`

Comment for the individual archive member as a *bytes* object.

`ZipInfo.extra`

Expansion field data. The [PKZIP Application Note](#) contains some comments on the internal structure of the data contained in this *bytes* object.

`ZipInfo.create_system`

Système ayant créé l'archive ZIP.

`ZipInfo.create_version`

Version de PKZIP ayant créé l'archive ZIP.

`ZipInfo.extract_version`

Version de PKZIP nécessaire à l'extraction de l'archive ZIP.

`ZipInfo.reserved`

Doit être à zéro.

`ZipInfo.flag_bits`

Bits d'options ZIP.

`ZipInfo.volume`

Numéro de volume de l'entête du fichier.

`ZipInfo.internal_attr`

Attributs internes.

`ZipInfo.external_attr`
Attributs de fichier externes.

`ZipInfo.header_offset`
Longueur de l'entête du fichier en octets.

`ZipInfo.CRC`
CRC-32 du fichier décompressé.

`ZipInfo.compress_size`
Taille des données décompressées.

`ZipInfo.file_size`
Taille du fichier décompressé.

13.5.4 Interface en ligne de commande

Le module `zipfile` fournit une interface en ligne de commande simple pour interagir avec des archives ZIP.

Si vous voulez créer une nouvelle archive ZIP, spécifiez son nom après l'option `-c` et listez ensuite le(s) nom(s) de fichier à inclure :

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Passer un répertoire est aussi possible :

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

Si vous voulez extraire une archive ZIP dans un répertoire donné, utilisez l'option `-e` :

```
$ python -m zipfile -e monty.zip target-dir/
```

Pour une liste des fichiers dans une archive ZIP, utilisez l'option `-l` :

```
$ python -m zipfile -l monty.zip
```

Options de la ligne de commande

- `-l <zipfile>`
Liste les fichiers dans un fichier ZIP *zipfile*.
- `-c <zipfile> <source1> ... <sourceN>`
Crée un fichier ZIP *zipfile* à partir des fichiers *source*.
- `-e <zipfile> <output_dir>`
Extrait le fichier ZIP *zipfile* vers le répertoire cible *output_dir*.
- `-t <zipfile>`
Teste si le fichier zip est valide.

13.6 tarfile — Read and write tar archive files

Code source : [Lib/tarfile.py](#)

The `tarfile` module makes it possible to read and write tar archives, including those using gzip, bz2 and lzma compression. Use the `zipfile` module to read or write .zip files, or the higher-level functions in `shutil`.

Some facts and figures :

- reads and writes `gzip`, `bz2` and `lzma` compressed archives if the respective modules are available.
- read/write support for the POSIX.1-1988 (ustar) format.
- read/write support for the GNU tar format including `longname` and `longlink` extensions, read-only support for all variants of the `sparse` extension including restoration of sparse files.
- read/write support for the POSIX.1-2001 (pax) format.
- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

Modifié dans la version 3.3 : Added support for `lzma` compression.

`tarfile.open` (*name=None, mode='r', fileobj=None, bufsize=10240, **kwargs*)

Return a `TarFile` object for the pathname *name*. For detailed information on `TarFile` objects and the keyword arguments that are allowed, see *TarFile Objects*.

mode has to be a string of the form `'filemode[:compression]'`, it defaults to `'r'`. Here is a full list of mode combinations :

mode	action
'r' ou 'r:*	Open for reading with transparent compression (recommended).
'r:'	Open for reading exclusively without compression.
'r:gz'	Open for reading with gzip compression.
'r:bz2'	Open for reading with bzip2 compression.
'r:xz'	Open for reading with lzma compression.
'x' ou 'x:'	Create a tarfile exclusively without compression. Raise an <code>FileExistsError</code> exception if it already exists.
'x:gz'	Create a tarfile with gzip compression. Raise an <code>FileExistsError</code> exception if it already exists.
'x:bz2'	Create a tarfile with bzip2 compression. Raise an <code>FileExistsError</code> exception if it already exists.
'x:xz'	Create a tarfile with lzma compression. Raise an <code>FileExistsError</code> exception if it already exists.
'a' ou 'a:'	Open for appending with no compression. The file is created if it does not exist.
'w' ou 'w:'	Open for uncompressed writing.
'w:gz'	Open for gzip compressed writing.
'w:bz2'	Open for bzip2 compressed writing.
'w:xz'	Open for lzma compressed writing.

Note that `'a:gz'`, `'a:bz2'` or `'a:xz'` is not possible. If *mode* is not suitable to open a certain (compressed) file for reading, `ReadError` is raised. Use *mode* `'r'` to avoid this. If a compression method is not supported, `CompressionError` is raised.

If *fileobj* is specified, it is used as an alternative to a *file object* opened in binary mode for *name*. It is supposed to be at position 0.

For modes `'w:gz'`, `'r:gz'`, `'w:bz2'`, `'r:bz2'`, `'x:gz'`, `'x:bz2'`, `tarfile.open()` accepts the keyword argument *compresslevel* (default 9) to specify the compression level of the file.

For special purposes, there is a second format for *mode* : 'filemode|[compression]'. `tarfile.open()` will return a `TarFile` object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, *fileobj* may be any object that has a `read()` or `write()` method (depending on the *mode*). *bufsize* specifies the blocksize and defaults to `20 * 512` bytes. Use this variant in combination with e.g. `sys.stdin`, a socket *file object* or a tape device. However, such a `TarFile` object is limited in that it does not allow random access, see *Examples*. The currently possible modes :

Mode	Action
'r *'	Open a <i>stream</i> of tar blocks for reading with transparent compression.
'r '	Open a <i>stream</i> of uncompressed tar blocks for reading.
'r gz'	Open a gzip compressed <i>stream</i> for reading.
'r bz2'	Open a bzip2 compressed <i>stream</i> for reading.
'r xz'	Open an lzma compressed <i>stream</i> for reading.
'w '	Open an uncompressed <i>stream</i> for writing.
'w gz'	Open a gzip compressed <i>stream</i> for writing.
'w bz2'	Open a bzip2 compressed <i>stream</i> for writing.
'w xz'	Open an lzma compressed <i>stream</i> for writing.

Modifié dans la version 3.5 : Le mode 'x' (création exclusive) est créé.

Modifié dans la version 3.6 : The *name* parameter accepts a *path-like object*.

class `tarfile.TarFile`

Class for reading and writing tar archives. Do not use this class directly : use `tarfile.open()` instead. See *TarFile Objects*.

`tarfile.is_tarfile(name)`

Return `True` if *name* is a tar archive file, that the `tarfile` module can read.

The `tarfile` module defines the following exceptions :

exception `tarfile.TarError`

Base class for all `tarfile` exceptions.

exception `tarfile.ReadError`

Is raised when a tar archive is opened, that either cannot be handled by the `tarfile` module or is somehow invalid.

exception `tarfile.CompressionError`

Is raised when a compression method is not supported or when the data cannot be decoded properly.

exception `tarfile.StreamError`

Is raised for the limitations that are typical for stream-like `TarFile` objects.

exception `tarfile.ExtractError`

Is raised for *non-fatal* errors when using `TarFile.extract()`, but only if `TarFile.errorlevel== 2`.

exception `tarfile.HeaderError`

Is raised by `TarInfo.frombuf()` if the buffer it gets is invalid.

The following constants are available at the module level :

`tarfile.ENCODING`

The default character encoding : 'utf-8' on Windows, the value returned by `sys.getfilesystemencoding()` otherwise.

Each of the following constants defines a tar archive format that the `tarfile` module is able to create. See section *Supported tar formats* for details.

`tarfile.USTAR_FORMAT`

POSIX.1-1988 (ustar) format.

`tarfile.GNU_FORMAT`

GNU tar format.

`tarfile.PAX_FORMAT`

POSIX.1-2001 (pax) format.

`tarfile.DEFAULT_FORMAT`

The default format for creating archives. This is currently `GNU_FORMAT`.

Voir aussi :

Module `zipfile` Documentation of the `zipfile` standard module.

Archiving operations Documentation of the higher-level archiving facilities provided by the standard `shutil` module.

GNU tar manual, Basic Tar Format Documentation for tar archive files, including GNU tar extensions.

13.6.1 TarFile Objects

The `TarFile` object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a `TarInfo` object, see *TarInfo Objects* for details.

A `TarFile` object can be used as a context manager in a `with` statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the *Examples* section for a use case.

Nouveau dans la version 3.2 : Added support for the context management protocol.

```
class tarfile.TarFile(name=None, mode='r', fileobj=None, format=DEFAULT_FORMAT,
                    tarinfo=TarInfo, dereference=False, ignore_zeros=False, encoding=ENCODING,
                    errors='surrogateescape', pax_headers=None, debug=0, errorlevel=0)
```

All following arguments are optional and can be accessed as instance attributes as well.

name is the pathname of the archive. *name* may be a *path-like object*. It can be omitted if *fileobj* is given. In this case, the file object's `name` attribute is used if it exists.

mode is either `'r'` to read from an existing archive, `'a'` to append data to an existing file, `'w'` to create a new file overwriting an existing one, or `'x'` to create a new file only if it does not already exist.

If *fileobj* is given, it is used for reading or writing data. If it can be determined, *mode* is overridden by *fileobj*'s *mode*. *fileobj* will be used from position 0.

Note : *fileobj* is not closed, when `TarFile` is closed.

format controls the archive format. It must be one of the constants `USTAR_FORMAT`, `GNU_FORMAT` or `PAX_FORMAT` that are defined at module level.

The *tarinfo* argument can be used to replace the default `TarInfo` class with a different one.

If *dereference* is `False`, add symbolic and hard links to the archive. If it is `True`, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If *ignore_zeros* is `False`, treat an empty block as the end of the archive. If it is `True`, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

debug can be set from 0 (no debug messages) up to 3 (all debug messages). The messages are written to `sys.stderr`.

If *errorlevel* is 0, all errors are ignored when using `TarFile.extract()`. Nevertheless, they appear as error messages in the debug output, when debugging is enabled. If 1, all *fatal* errors are raised as `OSError` exceptions.

If 2, all *non-fatal* errors are raised as `TarError` exceptions as well.

The *encoding* and *errors* arguments define the character encoding to be used for reading or writing the archive and how conversion errors are going to be handled. The default settings will work for most users. See section [Unicode issues](#) for in-depth information.

The *pax_headers* argument is an optional dictionary of strings which will be added as a pax global header if *format* is *PAX_FORMAT*.

Modifié dans la version 3.2 : Use 'surrogateescape' as the default for the *errors* argument.

Modifié dans la version 3.5 : Le mode 'x' (création exclusive) est créé.

Modifié dans la version 3.6 : The *name* parameter accepts a *path-like object*.

classmethod `TarFile.open(...)`

Alternative constructor. The `tarfile.open()` function is actually a shortcut to this classmethod.

`TarFile.getmember(name)`

Return a `TarInfo` object for member *name*. If *name* can not be found in the archive, `KeyError` is raised.

Note : If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

`TarFile.getmembers()`

Return the members of the archive as a list of `TarInfo` objects. The list has the same order as the members in the archive.

`TarFile.getnames()`

Return the members as a list of their names. It has the same order as the list returned by `getmembers()`.

`TarFile.list(verbose=True, *, members=None)`

Print a table of contents to `sys.stdout`. If *verbose* is *False*, only the names of the members are printed. If it is *True*, output similar to that of `ls -l` is produced. If optional *members* is given, it must be a subset of the list returned by `getmembers()`.

Modifié dans la version 3.5 : Added the *members* parameter.

`TarFile.next()`

Return the next member of the archive as a `TarInfo` object, when `TarFile` is opened for reading. Return *None* if there is no more available.

`TarFile.extractall(path=".", members=None, *, numeric_owner=False)`

Extract all members from the archive to the current working directory or directory *path*. If optional *members* is given, it must be a subset of the list returned by `getmembers()`. Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems : A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail.

If *numeric_owner* is *True*, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

Avertissement : Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `" / "` or filenames with two dots `" . . "`.

Modifié dans la version 3.5 : Added the *numeric_owner* parameter.

Modifié dans la version 3.6 : Le paramètre *path* accepte un objet chemin-compatible *path-like object*.

`TarFile.extract(member, path="", set_attrs=True, *, numeric_owner=False)`

Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. *member* may be a filename or a `TarInfo` object. You can specify a different directory using *path*. *path* may be a *path-like object*. File attributes (owner, mtime, mode) are set unless *set_attrs* is false.

If `numeric_owner` is `True`, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

Note : The `extract()` method does not take care of several extraction issues. In most cases you should consider using the `extractall()` method.

Avertissement : See the warning for `extractall()`.

Modifié dans la version 3.2 : Added the `set_attrs` parameter.

Modifié dans la version 3.5 : Added the `numeric_owner` parameter.

Modifié dans la version 3.6 : Le paramètre `path` accepte un objet chemin-compatible *path-like object*.

`TarFile.extractfile(member)`

Extract a member from the archive as a file object. *member* may be a filename or a `TarInfo` object. If *member* is a regular file or a link, an `io.BufferedReader` object is returned. Otherwise, `None` is returned.

Modifié dans la version 3.3 : Return an `io.BufferedReader` object.

`TarFile.add(name, arcname=None, recursive=True, exclude=None, *, filter=None)`

Add the file *name* to the archive. *name* may be any type of file (directory, fifo, symbolic link, etc.). If given, *arcname* specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting *recursive* to `False`. If *exclude* is given, it must be a function that takes one filename argument and returns a boolean value. Depending on this value the respective file is either excluded (`True`) or added (`False`). If *filter* is specified it must be a keyword argument. It should be a function that takes a `TarInfo` object argument and returns the changed `TarInfo` object. If it instead returns `None` the `TarInfo` object will be excluded from the archive. See *Examples* for an example.

Modifié dans la version 3.2 : Added the *filter* parameter.

Obsolète depuis la version 3.2 : The *exclude* parameter is deprecated, please use the *filter* parameter instead.

`TarFile.addfile(tarinfo, fileobj=None)`

Add the `TarInfo` object *tarinfo* to the archive. If *fileobj* is given, it should be a *binary file*, and *tarinfo.size* bytes are read from it and added to the archive. You can create `TarInfo` objects directly, or by using `gettinfo()`.

`TarFile.gettarinfo(name=None, arcname=None, fileobj=None)`

Create a `TarInfo` object from the result of `os.stat()` or equivalent on an existing file. The file is either named by *name*, or specified as a *file object* *fileobj* with a file descriptor. *name* may be a *path-like object*. If given, *arcname* specifies an alternative name for the file in the archive, otherwise, the name is taken from *fileobj*'s *name* attribute, or the *name* argument. The name should be a text string.

You can modify some of the `TarInfo`'s attributes before you add it using `addfile()`. If the file object is not an ordinary file object positioned at the beginning of the file, attributes such as *size* may need modifying. This is the case for objects such as `GzipFile`. The *name* may also be modified, in which case *arcname* could be a dummy string.

Modifié dans la version 3.6 : The *name* parameter accepts a *path-like object*.

`TarFile.close()`

Close the `TarFile`. In write mode, two finishing zero blocks are appended to the archive.

`TarFile.pax_headers`

A dictionary containing key-value pairs of pax global headers.

13.6.2 TarInfo Objects

A *TarInfo* object represents one member in a *TarFile*. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does *not* contain the file's data itself.

TarInfo objects are returned by *TarFile*'s methods `getmember()`, `getmembers()` and `gettarinfo()`.

class `tarfile.TarInfo` (*name=""*)

Create a *TarInfo* object.

classmethod `TarInfo.frombuf` (*buf*, *encoding*, *errors*)

Create and return a *TarInfo* object from string buffer *buf*.

Raises *HeaderError* if the buffer is invalid.

classmethod `TarInfo.fromtarfile` (*tarfile*)

Read the next member from the *TarFile* object *tarfile* and return it as a *TarInfo* object.

`TarInfo.tobuf` (*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)

Create a string buffer from a *TarInfo* object. For information on the arguments see the constructor of the *TarFile* class.

Modifié dans la version 3.2 : Use 'surrogateescape' as the default for the *errors* argument.

A *TarInfo* object has the following public data attributes :

`TarInfo.name`

Name of the archive member.

`TarInfo.size`

Size in bytes.

`TarInfo.mtime`

Time of last modification.

`TarInfo.mode`

Permission bits.

`TarInfo.type`

File type. *type* is usually one of these constants : `REGTYPE`, `AREGTYPE`, `LNKTYPE`, `SYMTYPE`, `DIRTYPE`, `FIFOTYPE`, `CONTTYPE`, `CHRTYPE`, `BLKTYPE`, `GNUTYPE_SPARSE`. To determine the type of a *TarInfo* object more conveniently, use the `is*()` methods below.

`TarInfo.linkname`

Name of the target file name, which is only present in *TarInfo* objects of type `LNKTYPE` and `SYMTYPE`.

`TarInfo.uid`

User ID of the user who originally stored this member.

`TarInfo.gid`

Group ID of the user who originally stored this member.

`TarInfo.uname`

User name.

`TarInfo.gname`

Group name.

`TarInfo.pax_headers`

A dictionary containing key-value pairs of an associated pax extended header.

A *TarInfo* object also provides some convenient query methods :


```
TarInfo.isfile()
    Return True if the Tarinfo object is a regular file.

TarInfo.isreg()
    Same as isfile().

TarInfo.isdir()
    Return True if it is a directory.

TarInfo.issym()
    Return True if it is a symbolic link.

TarInfo.islnk()
    Return True if it is a hard link.

TarInfo.ischr()
    Return True if it is a character device.

TarInfo.isblk()
    Return True if it is a block device.

TarInfo.isfifo()
    Return True if it is a FIFO.

TarInfo.isdev()
    Return True if it is one of character device, block device or FIFO.
```

13.6.3 Interface en ligne de commande

Nouveau dans la version 3.4.

The *tarfile* module provides a simple command-line interface to interact with tar archives.

If you want to create a new tar archive, specify its name after the *-c* option and then list the filename(s) that should be included :

```
$ python -m tarfile -c monty.tar spam.txt eggs.txt
```

Passer un répertoire est aussi possible :

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

If you want to extract a tar archive into the current directory, use the *-e* option :

```
$ python -m tarfile -e monty.tar
```

You can also extract a tar archive into a different directory by passing the directory's name :

```
$ python -m tarfile -e monty.tar other-dir/
```

For a list of the files in a tar archive, use the *-l* option :

```
$ python -m tarfile -l monty.tar
```

Options de la ligne de commande

```
-l <tarfile>
--list <tarfile>
    List files in a tarfile.

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
    Create tarfile from source files.

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]
    Extract tarfile into the current directory if output_dir is not specified.

-t <tarfile>
--test <tarfile>
    Test whether the tarfile is valid or not.

-v, --verbose
    Verbose output.
```

13.6.4 Exemples

How to extract an entire tar archive to the current working directory :

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

How to extract a subset of a tar archive with `TarFile.extractall()` using a generator function instead of a list :

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

How to create an uncompressed tar archive from a list of filenames :

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

The same example using the `with` statement :

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
```

(suite sur la page suivante)

(suite de la page précédente)

```
for name in ["foo", "bar", "quux"]:
    tar.add(name)
```

How to read a gzip compressed tar archive and display some member information :

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

How to create an archive and reset the user information using the *filter* parameter in *TarFile.add()* :

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.6.5 Supported tar formats

There are three tar formats that can be created with the *tarfile* module :

- The POSIX.1-1988 ustar format (*USTAR_FORMAT*). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 GiB. This is an old and limited but widely supported format.
- The GNU tar format (*GNU_FORMAT*). It supports long filenames and linknames, files bigger than 8 GiB and sparse files. It is the de facto standard on GNU/Linux systems. *tarfile* fully supports the GNU tar extensions for long names, sparse file support is read-only.
- The POSIX.1-2001 pax format (*PAX_FORMAT*). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. However, not all tar implementations today are able to handle pax archives properly.
The *pax* format is an extension to the existing *ustar* format. It uses extra headers for information that cannot be stored otherwise. There are two flavours of pax headers : Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in *UTF-8* for portability reasons.

There are some more variants of the tar format which can be read, but not created :

- The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.
- The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.

13.6.6 Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (which is the basis of all other formats) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a *UTF-8* system cannot be read correctly on a *Latin-1* system if it contains non-*ASCII* characters. Textual metadata (like filenames, linknames, user/group names) will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive. The pax format was designed to solve this problem. It stores non-*ASCII* metadata using the universal character encoding *UTF-8*.

The details of character conversion in *tarfile* are controlled by the *encoding* and *errors* keyword arguments of the *TarFile* class.

encoding defines the character encoding to use for the metadata in the archive. The default value is *sys.getfilesystemencoding()* or *'ascii'* as a fallback. Depending on whether the archive is read or written, the metadata must be either decoded or encoded. If *encoding* is not set appropriately, this conversion may fail.

The *errors* argument defines how characters are treated that cannot be converted. Possible values are listed in section *Gestionnaires d'erreurs*. The default scheme is *'surrogateescape'* which Python also uses for its file system calls, see *Noms de fichiers, arguments en ligne de commande, et variables d'environnement*.

In case of *PAX_FORMAT* archives, *encoding* is generally not needed because all the metadata is stored using *UTF-8*. *encoding* is only used in the rare cases when binary pax headers are decoded or when strings with surrogate characters are stored.

Les modules décrits dans ce chapitre lisent divers formats de fichier qui ne sont ni des langages balisés ni relatifs aux e-mails.

14.1 `csv` — Lecture et écriture de fichiers CSV

Code source : [Lib/csv.py](#)

Le format CSV (*Comma Separated Values*, valeurs séparées par des virgules) est le format le plus commun dans l'importation et l'exportation de feuilles de calculs et de bases de données. Le format fut utilisé pendant des années avant qu'aient lieu des tentatives de standardisation avec la **RFC 4180**. L'absence de format bien défini signifie que des différences subtiles existent dans la production et la consommation de données par différentes applications. Ces différences peuvent gêner lors du traitement de fichiers CSV depuis des sources multiples. Cependant, bien que les séparateurs et délimiteurs varient, le format global est suffisamment similaire pour qu'un module unique puisse manipuler efficacement ces données, masquant au programmeur les détails de lecture/écriture des données.

Le module `csv` implémente des classes pour lire et écrire des données tabulaires au format CSV. Il vous permet de dire « écris ces données dans le format préféré par Excel » ou « lis les données de ce fichier généré par Excel », sans connaître les détails précis du format CSV utilisé par Excel. Vous pouvez aussi décrire les formats CSV utilisés par d'autres application ou définir vos propres spécialisations.

Les objets `reader` et `writer` du module `csv` lisent et écrivent des séquences. Vous pouvez aussi lire/écrire les données dans un dictionnaire en utilisant les classes `DictReader` et `DictWriter`.

Voir aussi :

PEP 305 — Interface des fichiers CSV La proposition d'amélioration de Python (PEP) qui a proposé cet ajout au langage.

14.1.1 Contenu du module

Le module `csv` définit les fonctions suivantes :

`csv.reader(csvfile, dialect='excel', **fmtparams)`

Renvoie un objet lecteur, qui itérera sur les lignes de l'objet `csvfile` donné. `csvfile` peut être n'importe quel objet supportant le protocole *itérateur* et renvoyant une chaîne de caractères chaque fois que sa méthode `__next__()` est appelée — les *fichiers objets* et les listes sont tous deux valables. Si `csvfile` est un fichier, il doit être ouvert avec `newline=''`.¹ Un paramètre `dialect` optionnel peut être fourni pour définir un ensemble de paramètres spécifiques à un dialecte CSV particulier. Il peut s'agir d'une instance de sous-classe de `Dialect` ou de l'une des chaînes renvoyées par la fonction `list_dialects()`. Les autres arguments nommés optionnels (`fmtparams`) peuvent être spécifiés pour redéfinir des paramètres de formatage particuliers dans le dialecte courant. Pour des détails complets sur les dialectes et paramètres de formatage, voir la section *Dialectes et paramètres de formatage*. Chaque ligne lue depuis le fichier CSV est renvoyée comme une liste de chaînes de caractères. Aucune conversion automatique de type des données n'est effectuée à moins que l'option de formatage `QUOTE_NONNUMERIC` soit spécifiée (dans ce cas, les champs sans guillemets sont transformés en nombres flottants).

Un court exemple d'utilisation :

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

`csv.writer(csvfile, dialect='excel', **fmtparams)`

Renvoie un objet transcritteur responsable de convertir les données de l'utilisateur en chaînes délimitées sur l'objet fichier-compatible donné. `csvfile` peut être n'importe quel objet avec une méthode `write()`. Si `csvfile` est un fichier, il doit être ouvert avec `newline=''`.¹ Un paramètre `dialect` optionnel peut être fourni pour définir un ensemble de paramètres spécifiques à un dialecte CSV particulier. Il peut s'agir d'une instance de sous-classe de `Dialect` ou de l'une des chaînes renvoyées par la fonction `list_dialects()`. Les autres arguments nommés optionnels (`fmtparams`) peuvent être spécifiés pour redéfinir des paramètres de formatage particuliers dans le dialecte courant. Pour des détails complets sur les dialectes et paramètres de formatage, voir la section *Dialectes et paramètres de formatage*. Pour faciliter au mieux l'interfaçage avec d'autres modules implémentant l'interface *DB*, la valeur `None` est écrite comme une chaîne vide. Bien que ce ne soit pas une transformation réversible, cela simplifie l'exportation de données SQL *NULL* vers des fichiers CSV sans pré-traiter les données renvoyées par un appel à `cursor.fetch*`. Toutes les autres données qui ne sont pas des chaînes de caractères sont transformées en chaînes par un appel à `str()` avant d'être écrites.

Un court exemple d'utilisation :

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                           quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

`csv.register_dialect(name[, dialect[, **fmtparams]])`

Associe `dialect` avec `name`. `name` doit être une chaîne de caractères. Le dialecte peut être spécifié en passant une instance d'une sous-classe de `Dialect`, des arguments nommés `fmtparams`, ou les deux, avec les arguments nommés redéfinissant les paramètres du dialecte. Pour des détails complets sur les dialectes et paramètres de formatage, voir la section *Dialectes et paramètres de formatage*.

1. Si `newline=''` n'est pas spécifié, les caractères de fin de ligne embarqués dans des champs délimités par des guillemets ne seront pas interprétés correctement, et sur les plateformes qui utilisent `\r\n` comme marqueur de fin de ligne, un `\r` sera ajouté. Vous devriez toujours spécifier sans crainte `newline=''`, puisque le module `csv` gère lui-même les fins de lignes (*universelles*).

`csv.unregister_dialect(name)`

Supprime le dialecte associé à *name* depuis le registre des dialectes. Une *Error* est levée si *name* n'est pas un nom de dialecte enregistré.

`csv.get_dialect(name)`

Renvoie le dialecte associé à *name*. Une *Error* est levée si *name* n'est pas un nom de dialecte enregistré. Cette fonction renvoie un objet *Dialect* immuable.

`csv.list_dialects()`

Renvoie les noms de tous les dialectes enregistrés.

`csv.field_size_limit([new_limit])`

Renvoie la taille de champ maximale courante autorisée par l'analyseur. Si *new_limit* est donnée, elle devient la nouvelle limite.

Le module *csv* définit les classes suivantes :

class `csv.DictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwargs)`

Crée un objet qui opère comme un lecteur ordinaire mais assemble les informations de chaque ligne dans un *OrderedDict* dont les clés sont données par le paramètre optionnel *fieldnames*.

Le paramètre *fieldnames* est une *sequence*. Si *fieldnames* est omis, les valeurs de la première ligne du fichier *f* seront utilisées comme noms de champs. Sans se soucier de comment sont déterminés les noms de champs, le dictionnaire ordonné préserve leur ordre original.

Si une ligne a plus de champs que *fieldnames*, les données excédentaires sont mises dans une liste stockée dans le champ spécifié par *restkey* (None par défaut). Si une ligne non-vide a moins de champs que *fieldnames*, les valeurs manquantes sont mises à None.

Tous les autres arguments optionnels ou nommés sont passés à l'instance *reader* sous-jacente.

Modifié dans la version 3.6 : Les lignes renvoyées sont maintenant de type *OrderedDict*.

Un court exemple d'utilisation :

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
OrderedDict([('first_name', 'John'), ('last_name', 'Cleese')])
```

class `csv.DictWriter(f, fieldnames, restval="", extrasaction='raise', dialect='excel', *args, **kwargs)`

Crée un objet qui opère comme un transcritteur ordinaire mais qui produit les lignes de sortie depuis des dictionnaires. Le paramètre *fieldnames* est une *séquence* de clés qui indique l'ordre dans lequel les valeurs du dictionnaire passé à la méthode *writerow()* doivent être écrites vers le fichier *f*. Le paramètre optionnel *restval* spécifie la valeur à écrire si une clé de *fieldnames* manque dans le dictionnaire. Si le dictionnaire passé à *writerow()* possède une clé non présente dans *fieldnames*, le paramètre optionnel *extrasaction* indique quelle action réaliser. S'il vaut 'raise', sa valeur par défaut, une *ValueError* est levée. S'il faut 'ignore', les valeurs excédentaires du dictionnaire sont ignorées. Les autres arguments optionnels ou nommés sont passés à l'instance *writer* sous-jacente.

Notez que contrairement à la classe *DictReader*, le paramètre *fieldnames* de *DictWriter* n'est pas optionnel. Puisque les objets *dict* de Python ne sont pas ordonnés, il n'y a pas d'information suffisante pour déduire l'ordre dans lequel la ligne devrait être transcrite vers le fichier *f*.

Un court exemple d'utilisation :

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class `csv.Dialect`

La classe *Dialect* est une classe de conteneurs utilisée principalement pour ses attributs, qui servent à définir des paramètres pour des instances spécifiques de *reader* ou *writer*.

class `csv.excel`

La classe *excel* définit les propriétés usuelles d'un fichier CSV généré par Excel. Elle est enregistrée avec le nom de dialecte 'excel'.

class `csv.excel_tab`

La classe *excel_tab* définit les propriétés usuelles d'un fichier CSV généré par Excel avec des tabulations comme séparateurs. Elle est enregistrée avec le nom de dialecte 'excel-tab'.

class `csv.unix_dialect`

La classe *unix_dialect* définit les propriétés usuelles d'un fichier CSV généré sur un système Unix, c'est-à-dire utilisant '\n' comme marqueur de fin de ligne et délimitant tous les champs par des guillemets. Elle est enregistrée avec le nom de dialecte 'unix'.

Nouveau dans la version 3.2.

class `csv.Sniffer`

La classe *Sniffer* est utilisée pour déduire le format d'un fichier CSV.

La classe *Sniffer* fournit deux méthodes :

sniff (*sample*, *delimiters=None*)

Analyse l'extrait donné (*sample*) et renvoie une sous-classe *Dialect* reflétant les paramètres trouvés. Si le paramètre optionnel *delimiters* est donné, il est interprété comme une chaîne contenant tous les caractères valides de séparation possibles.

has_header (*sample*)

Analyse l'extrait de texte (présupposé être au format CSV) et renvoie *True* si la première ligne semble être une série d'en-têtes de colonnes.

Un exemple d'utilisation de *Sniffer* :

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

Le module *csv* définit les constantes suivantes :

csv.QUOTE_ALL

Indique aux objets *writer* de délimiter tous les champs par des guillemets.

csv.QUOTE_MINIMAL

Indique aux objets *writer* de ne délimiter ainsi que les champs contenant un caractère spécial comme *delimiter*, *quotechar* ou n'importe quel caractère de *lineterminator*.

csv.QUOTE_NONNUMERIC

Indique aux objets *writer* de délimiter ainsi tous les champs non-numériques.

Indique au lecteur de convertir tous les champs non délimités par des guillemets vers des *float*.

csv.QUOTE_NONE

Indique aux objets *writer* de ne jamais délimiter les champs par des guillemets. Quand le *delimiter* courant apparaît dans les données, il est précédé sur la sortie par un caractère *escapechar*. Si *escapechar* n'est pas précisé, le transcritteur lèvera une *Error* si un caractère nécessitant un échappement est rencontré.

Indique au *reader* de ne pas opérer de traitement spécial sur les guillemets.

Le module *csv* définit les exceptions suivantes :

exception csv.Error

Levée par les fonctions du module quand une erreur détectée.

14.1.2 Dialectes et paramètres de formatage

Pour faciliter la spécification du format des entrées et sorties, les paramètres de formatage spécifiques sont regroupés en dialectes. Un dialecte est une sous-classe de *Dialect* avec un ensemble de méthodes spécifiques et une méthode *validate()*. Quand un objet *reader* ou *writer* est créé, vous pouvez spécifier une chaîne ou une sous-classe de *Dialect* comme paramètre *dialect*. En plus du paramètre *dialect*, ou à sa place, vous pouvez aussi préciser des paramètres de formatage individuels, qui ont les mêmes noms que les attributs de *Dialect* définis ci-dessous.

Les dialectes supportent les attributs suivants :

Dialect.delimiter

Une chaîne d'un seul caractère utilisée pour séparer les champs. Elle vaut *' , '* par défaut.

Dialect.doublequote

Contrôle comment les caractères *quotechar* dans le champ doivent être retranscrits. Quand ce paramètre vaut *True*, le caractère est doublé. Quand il vaut *False*, le caractère *escapechar* est utilisé comme préfixe à *quotechar*. Il vaut *True* par défaut.

En écriture, si *doublequote* vaut *False* et qu'aucun *escapechar* n'est précisé, une *Error* est levée si un *quotechar* est trouvé dans le champ.

Dialect.escapechar

Une chaîne d'un seul caractère utilisée par le transcritteur pour échapper *delimiter* si *quoting* vaut *QUOTE_NONE*, et pour échapper *quotechar* si *doublequote* vaut *False*. À la lecture, *escapechar* retire toute signification spéciale au caractère qui le suit. Elle vaut par défaut *None*, ce qui désactive l'échappement.

Dialect.lineterminator

La chaîne utilisée pour terminer les lignes produites par un *writer*. Elle vaut par défaut *'\r\n'*.

Note : La classe *reader* est codée en dur pour reconnaître *'\r'* et *'\n'* comme marqueurs de fin de ligne, et ignorer *lineterminator*. Ce comportement pourrait changer dans le futur.

Dialect.quotechar

Une chaîne d'un seul caractère utilisée pour délimiter les champs contenant des caractères spéciaux, comme *delimiter* ou *quotechar*, ou contenant un caractère de fin de ligne. Elle vaut *'\"'* par défaut.

Dialect.quoting

Contrôle quand les guillemets doivent être générés par le transcritteur et reconnus par le lecteur. Il peut prendre comme valeur l'une des constantes *QUOTE_** (voir la section *Contenu du module*) et vaut par défaut *QUOTE_MINIMAL*.

Dialect.skipinitialspace

Quand il vaut *True*, les espaces suivant directement *delimiter* sont ignorés. Il vaut *False* par défaut.

`Dialect.strict`

Quand il vaut `True`, une exception `Error` est levée lors de mauvaises entrées CSV. Il vaut `False` par défaut.

14.1.3 Objets lecteurs

Les objets lecteurs (instances de `DictReader` ou objets renvoyés par la fonction `reader()`) ont les méthodes publiques suivantes :

`csvreader.__next__()`

Renvoie la ligne suivante de l'objet itérable du lecteur en tant que liste (si l'objet est renvoyé depuis `reader()`) ou dictionnaire (si l'objet est un `DictReader`), analysé suivant le dialecte courant. Généralement, vous devez appeler la méthode à l'aide de `next(reader)`.

Les objets lecteurs ont les attributs publics suivants :

`csvreader.dialect`

Une description en lecture seule du dialecte utilisé par l'analyseur.

`csvreader.line_num`

Le nombre de lignes lues depuis l'itérateur source. Ce n'est pas équivalent au nombre d'enregistrements renvoyés, puisque certains enregistrements peuvent s'étendre sur plusieurs lignes.

Les objets `DictReader` ont les attributs publics suivants :

`csvreader.fieldnames`

S'il n'est pas passé comme paramètre à la création de l'objet, cet attribut est initialisé lors du premier accès ou quand le premier enregistrement est lu depuis le fichier.

14.1.4 Objets transcripteurs

Les objets `Writer` (instances de `DictWriter` ou objets renvoyés par la fonction `writer()`) ont les méthodes publiques suivantes. Une `row` doit être un itérable de chaînes de caractères ou de nombres pour les objets `Writer`, et un dictionnaire associant des noms de champs à des chaînes ou des nombres (en les faisant d'abord passer par `str()`) pour les objets `DictWriter`. Notez que les nombres complexes sont retranscrits entourés de parenthèses. Cela peut causer quelques problèmes pour d'autres programmes qui liraient ces fichiers CSV (en supposant qu'ils supportent les nombres complexes).

`csvwriter.writerow(row)`

Écrit le paramètre `row` vers le fichier associé au transcripteur, formaté selon le dialecte courant.

Modifié dans la version 3.5 : Ajout du support d'itérables arbitraires.

`csvwriter.writerows(rows)`

Écrit tous les éléments de `rows` (itérable d'objets `row` comme décrits précédemment) vers le fichier associé au transcripteur, formatés selon le dialecte courant.

Les objets transcripteurs ont les attributs publics suivants :

`csvwriter.dialect`

Une description en lecture seule du dialecte utilisé par le transcripteur.

Les objets `DictWriter` ont les attributs publics suivants :

`DictWriter.writeheader()`

Écrit une ligne contenant les noms de champs (comme spécifiés au constructeur).

Nouveau dans la version 3.2.

14.1.5 Exemples

Le plus simple exemple de lecture d'un fichier CSV :

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Lire un fichier avec un format alternatif :

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

Le plus simple exemple d'écriture correspondant est :

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Puisque `open()` est utilisée pour ouvrir un fichier CSV en lecture, le fichier sera par défaut décodé vers Unicode en utilisant l'encodage par défaut (voir `locale.getpreferredencoding()`). Pour décoder un fichier utilisant un encodage différent, utilisez l'argument `encoding` de `open` :

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

La même chose s'applique lors de l'écriture dans un autre encodage que celui par défaut du système : spécifiez l'encodage en argument lors de l'ouverture du fichier de sortie.

Enregistrer un nouveau dialecte :

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

Un exemple d'utilisation un peu plus avancé du lecteur — attrapant et notifiant les erreurs :

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

Et bien que le module ne permette pas d'analyser directement des chaînes, cela peut être fait facilement :

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

Notes

14.2 configparser — Configuration file parser

Code source : [Lib/configparser.py](#)

This module provides the `ConfigParser` class which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

Note : This library does *not* interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

Voir aussi :

Module `shlex` Support for creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.

Module `json` The json module implements a subset of JavaScript syntax which can also be used for this purpose.

14.2.1 Quick Start

Let's take a very basic configuration file that looks like this :

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

The structure of INI files is described *in the following section*. Essentially, the file consists of sections, each of which contains keys with values. `configparser` classes can read and write such files. Let's start by creating the above configuration file programmatically.

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                    'Compression': 'yes',
...                    'CompressionLevel': '9'}
```

(suite sur la page suivante)

(suite de la page précédente)

```

>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
>>> topsecret['Port'] = '50022'      # mutates the parser
>>> topsecret['ForwardX11'] = 'no'   # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...     config.write(configfile)
...

```

As you can see, we can treat a config parser much like a dictionary. There are differences, *outlined later*, but the behavior is very close to what you would expect from a dictionary.

Now that we have created and saved a configuration file, let's read it back and explore the data it holds.

```

>>> import configparser
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']: print(key)
...
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['bitbucket.org']['ForwardX11']
'yes'

```

As we can see above, the API is pretty straightforward. The only bit of magic involves the `DEFAULT` section which provides default values for all other sections¹. Note also that keys in sections are case-insensitive and stored in lowercase¹.

1. Config parsers allow for heavy customization. If you are interested in changing the behaviour outlined by the footnote reference, consult the *Customizing Parser Behaviour* section.

14.2.2 Supported Datatypes

Config parsers do not guess datatypes of values in configuration files, always storing them internally as strings. This means that if you need other datatypes, you should convert on your own :

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

Since this task is so common, config parsers provide a range of handy getter methods to handle integers, floats and booleans. The last one is the most interesting because simply passing the value to `bool()` would do no good since `bool('False')` is still `True`. This is why config parsers also provide `getboolean()`. This method is case-insensitive and recognizes Boolean values from 'yes'/'no', 'on'/'off', 'true'/'false' and '1'/'0'¹. For example :

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['bitbucket.org'].getboolean('ForwardX11')
True
>>> config.getboolean('bitbucket.org', 'Compression')
True
```

Apart from `getboolean()`, config parsers also provide equivalent `getint()` and `getfloat()` methods. You can register your own converters and customize the provided ones.¹

14.2.3 Fallback Values

As with a dictionary, you can use a section's `get()` method to provide fallback values :

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

Please note that default values have precedence over fallback values. For instance, in our example the 'CompressionLevel' key was specified only in the 'DEFAULT' section. If we try to get it from the section 'topsecret.server.com', we will always get the default, even if we specify a fallback :

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

One more thing to be aware of is that the parser-level `get()` method provides a custom, more complex interface, maintained for backwards compatibility. When using this method, a fallback value can be provided via the `fallback` keyword-only argument :

```
>>> config.get('bitbucket.org', 'monster',
...           fallback='No such things as monsters')
'No such things as monsters'
```

The same `fallback` argument can be used with the `getint()`, `getfloat()` and `getboolean()` methods, for example :

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 Supported INI File Structure

A configuration file consists of sections, each led by a `[section]` header, followed by key/value entries separated by a specific string (= or : by default¹). By default, section names are case sensitive but keys are not¹. Leading and trailing whitespace is removed from keys and values. Values can be omitted, in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser's mode, blank lines may be treated as parts of multiline values or ignored.

Configuration files may include comments, prefixed by specific characters (# and ; by default¹). Comments may appear on their own on an otherwise empty line, possibly indented.¹

Par exemple :

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
      I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
```

(suite sur la page suivante)

(suite de la page précédente)

```
multiline_values = are
    handled just fine as
    long as they are indented
    deeper than the first line
    of a value
# Did I mention we can indent comments, too?
```

14.2.5 Interpolation of values

On top of the core functionality, *ConfigParser* supports interpolation. This means values can be preprocessed before returning them from `get()` calls.

class configparser.BasicInterpolation

The default implementation used by *ConfigParser*. It enables values to contain format strings which refer to other values in the same section, or values in the special default section¹. Additional default values can be provided on initialization.

Par exemple :

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures
```

In the example above, *ConfigParser* with *interpolation* set to `BasicInterpolation()` would resolve `%(home_dir)s` to the value of `home_dir` (`/Users` in this case). `%(my_dir)s` in effect would resolve to `/Users/lumberjack`. All interpolations are done on demand so keys used in the chain of references do not have to be specified in any specific order in the configuration file.

With interpolation set to `None`, the parser would simply return `%(my_dir)s/Pictures` as the value of `my_pictures` and `%(home_dir)s/lumberjack` as the value of `my_dir`.

class configparser.ExtendedInterpolation

An alternative handler for interpolation which implements a more advanced syntax, used for instance in `zc.buildout`. Extended interpolation is using `${section:option}` to denote a value from a foreign section. Interpolation can span multiple levels. For convenience, if the `section:` part is omitted, interpolation defaults to the current section (and possibly the default values from the special section).

For example, the configuration specified above with basic interpolation, would look like this with extended interpolation :

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures
```

Values from other sections can be fetched as well :

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/
```

(suite sur la page suivante)

(suite de la page précédente)

```
[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.6 Mapping Protocol Access

Nouveau dans la version 3.2.

Mapping protocol access is a generic name for functionality that enables using custom objects as if they were dictionaries. In case of *configparser*, the mapping interface implementation is using the `parser['section']['option']` notation.

`parser['section']` in particular returns a proxy for the section's data in the parser. This means that the values are not copied but they are taken from the original parser on demand. What's even more important is that when values are changed on a section proxy, they are actually mutated in the original parser.

configparser objects behave as close to actual dictionaries as possible. The mapping interface is complete and adheres to the *MutableMapping* ABC. However, there are a few differences that should be taken into account :

- By default, all keys in sections are accessible in a case-insensitive manner¹. E.g. for `option` in `parser["section"]` yields only `optionxform`'ed option key names. This means lowercased keys by default. At the same time, for a section that holds the key 'a', both expressions return `True` :

```
"a" in parser["section"]
"A" in parser["section"]
```

- All sections include `DEFAULTSECT` values as well which means that `.clear()` on a section may not leave the section visibly empty. This is because default values cannot be deleted from the section (because technically they are not there). If they are overridden in the section, deleting causes the default value to be visible again. Trying to delete a default value causes a `KeyError`.
- `DEFAULTSECT` cannot be removed from the parser :
 - trying to delete it raises `ValueError`,
 - `parser.clear()` leaves it intact,
 - `parser.popitem()` never returns it.
- `parser.get(section, option, **kwargs)` - the second argument is **not** a fallback value. Note however that the section-level `get()` methods are compatible both with the mapping protocol and the classic *configparser* API.
- `parser.items()` is compatible with the mapping protocol (returns a list of *section_name*, *section_proxy* pairs including the `DEFAULTSECT`). However, this method can also be invoked with arguments : `parser.items(section, raw, vars)`. The latter call returns a list of *option*, *value* pairs for a specified section, with all interpolations expanded (unless `raw=True` is provided).

The mapping protocol is implemented on top of the existing legacy API so that subclasses overriding the original interface still should have mappings working as expected.

14.2.7 Customizing Parser Behaviour

There are nearly as many INI format variants as there are applications using it. `configparser` goes a long way to provide support for the largest sensible set of INI styles available. The default functionality is mainly dictated by historical background and it's very likely that you will want to customize some of the features.

The most common way to change the way a specific config parser works is to use the `__init__()` options :

- `defaults`, default value : `None`

This option accepts a dictionary of key-value pairs which will be initially put in the `DEFAULT` section. This makes for an elegant way to support concise configuration files that don't specify values which are the same as the documented default.

Hint : if you want to specify default values for a specific section, use `read_dict()` before you read the actual file.

- `dict_type`, default value : `collections.OrderedDict`

This option has a major impact on how the mapping protocol will behave and how the written configuration files look. With the default ordered dictionary, every section is stored in the order they were added to the parser. Same goes for options within sections.

An alternative dictionary type can be used for example to sort sections and options on write-back. You can also use a regular dictionary for performance reasons.

Please note : there are ways to add a set of key-value pairs in a single operation. When you use a regular dictionary in those operations, the order of the keys may be random. For example :

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                               'key2': 'value2',
...                               'key3': 'value3'},
...                  'section2': {'keyA': 'valueA',
...                               'keyB': 'valueB',
...                               'keyC': 'valueC'},
...                  'section3': {'foo': 'x',
...                               'bar': 'y',
...                               'baz': 'z'}})
>>> parser.sections()
['section3', 'section2', 'section1']
>>> [option for option in parser['section3']]
['baz', 'foo', 'bar']
```

In these operations you need to use an ordered dictionary as well :

```
>>> from collections import OrderedDict
>>> parser = configparser.ConfigParser()
>>> parser.read_dict(
...     OrderedDict((
...         ('s1',
...          OrderedDict((
...              ('1', '2'),
...              ('3', '4'),
...              ('5', '6'),
...          ))),
...     ),
...     ('s2',
...      OrderedDict((
...          ('a', 'b'),
...          ('c', 'd'),
...          ('e', 'f'),
...      )))
```

(suite sur la page suivante)

(suite de la page précédente)

```

...     ),
...     ))
... )
>>> parser.sections()
['s1', 's2']
>>> [option for option in parser['s1']]
['1', '3', '5']
>>> [option for option in parser['s2'].values()]
['b', 'd', 'f']

```

- `allow_no_value`, default value : `False`

Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by `configparser`. The `allow_no_value` parameter to the constructor can be used to indicate that such values should be accepted :

```

>>> import configparser

>>> sample_config = """
... [mysqld]
...     user = mysql
...     pid-file = /var/run/mysqld/mysqld.pid
...     skip-external-locking
...     old_passwords = 1
...     skip-bdb
...     # we don't need ACID today
...     skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]
None

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
...
KeyError: 'does-not-exist'

```

- `delimiters`, default value : `('=', ':')`

Delimiters are substrings that delimit keys from values within a section. The first occurrence of a delimiting substring on a line is considered a delimiter. This means values (but not keys) can contain the delimiters.

See also the `space_around_delimiters` argument to `ConfigParser.write()`.

- `comment_prefixes`, default value : `(';', '#')`

- `inline_comment_prefixes`, default value : `None`

Comment prefixes are strings that indicate the start of a valid comment within a config file. `comment_prefixes` are used only on otherwise empty lines (optionally indented) whereas `inline_comment_prefixes` can be used after every valid value (e.g. section names, options and empty lines as well). By default inline comments are disabled and `#` and `;` are used as prefixes for whole line comments.

Modifié dans la version 3.2 : In previous versions of `configparser` behaviour matched `comment_prefixes=(';', '#')` and `inline_comment_prefixes=(';',)`.

Please note that config parsers don't support escaping of comment prefixes so using `inline_comment_prefixes` may prevent users from specifying option values with characters used as comment prefixes. When in doubt, avoid

setting `inline_comment_prefixes`. In any circumstances, the only way of storing comment prefix characters at the beginning of a line in multiline values is to interpolate the prefix, for example :

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...     ${hash}!/usr/bin/env python
...     ${hash} -*- coding: utf-8 -*-
...
... extensions =
...     enabled_extension
...     another_extension
...     #disabled_by_comment
...     yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...     line #2
...     line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- `strict`, default value : True

When set to True, the parser will not allow for any section or option duplicates while reading from a single source (using `read_file()`, `read_string()` or `read_dict()`). It is recommended to use strict parsers in new applications.

Modifié dans la version 3.2 : In previous versions of `configparser` behaviour matched `strict=False`.

- `empty_lines_in_values`, default value : True

In config parsers, values can span multiple lines as long as they are indented more than the key that holds them. By default parsers also let empty lines to be parts of values. At the same time, keys can be arbitrarily indented themselves to improve readability. In consequence, when configuration files get big and complex, it is easy for the user to lose track of the file structure. Take for instance :

```
[Section]
key = multiline
    value with a gotcha

    this = is still a part of the multiline value of 'key'
```


(suite de la page précédente)

```

... [Section2]
... AnotherKey = Value
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']

```

ConfigParser.SECTCRE

A compiled regular expression used to parse section headers. The default matches `[section]` to the name "section". Whitespace is considered part of the section name, thus `[larch]` will be read as a section of name " larch ". Override this attribute if that's unsuitable. For example :

```

>>> config = """
... [Section 1]
... option = value
...
... [ Section 2 ]
... another = val
... """
>>> typical = ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', ' Section 2 ']
>>> custom = ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^\]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']

```

Note : While ConfigParser objects also use an OPTCRE attribute for recognizing option lines, it's not recommended to override it because that would interfere with constructor options *allow_no_value* and *delimiters*.

14.2.8 Legacy API Examples

Mainly because of backwards compatibility concerns, `configparser` provides also a legacy API with explicit get/set methods. While there are valid use cases for the methods outlined below, mapping protocol access is preferred for new projects. The legacy API is at times more advanced, low-level and downright counterintuitive.

An example of writing to a configuration file :

```

import configparser

config = configparser.RawConfigParser()

```

(suite sur la page suivante)

(suite de la page précédente)

```
# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again :

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

To get interpolation, use `ConfigParser` :

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False)) # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))  # -> "%(bar)s is %(baz)s!"

# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
# -> "Python is fun!"
```

(suite sur la page suivante)

(suite de la page précédente)

```

print(cfg.get('Section1', 'foo', fallback='Monty is not.))
    # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.))
    # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
    # -> None

```

Default values are available in both types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```

import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))      # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))      # -> "Life is hard!"

```

14.2.9 ConfigParser Objects

class configparser.ConfigParser (defaults=None, dict_type=collections.OrderedDict, allow_no_value=False, delimiters=('=', ':'), comment_prefixes=(';', '#'), inline_comment_prefixes=None, strict=True, empty_lines_in_values=True, default_section=configparser.DEFAULTSECT, interpolation=BasicInterpolation(), converters={})

The main configuration parser. When *defaults* is given, it is initialized into the dictionary of intrinsic defaults. When *dict_type* is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values.

When *delimiters* is given, it is used as the set of substrings that divide keys from values. When *comment_prefixes* is given, it will be used as the set of substrings that prefix comments in otherwise empty lines. Comments can be indented. When *inline_comment_prefixes* is given, it will be used as the set of substrings that prefix comments in non-empty lines.

When *strict* is True (the default), the parser won't allow for any section or option duplicates while reading from a single source (file, string or dictionary), raising *DuplicateSectionError* or *DuplicateOptionError*. When *empty_lines_in_values* is False (default: True), each empty line marks the end of an option. Otherwise, internal empty lines of a multiline option are kept as part of the value. When *allow_no_value* is True (default: False), options without values are accepted; the value held for these is None and they are serialized without the trailing delimiter.

When *default_section* is given, it specifies the name for the special section holding default values for other sections and interpolation purposes (normally named "DEFAULT"). This value can be retrieved and changed on runtime using the *default_section* instance attribute.

Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. None can be used to turn off interpolation completely, *ExtendedInterpolation()* provides a more advanced variant inspired by *zc.buildout*. More on the subject in the [dedicated documentation section](#).

All option names used in interpolation will be passed through the `optionxform()` method just like any other option name reference. For example, using the default implementation of `optionxform()` (which converts option names to lower case), the values `foo %(bar)s` and `foo %(BAR)s` are equivalent.

When `converters` is given, it should be a dictionary where each key represents the name of a type converter and each value is a callable implementing the conversion from string to the desired datatype. Every converter gets its own corresponding `get*()` method on the parser object and section proxies.

Modifié dans la version 3.1 : The default `dict_type` is `collections.OrderedDict`.

Modifié dans la version 3.2 : `allow_no_value`, `delimiters`, `comment_prefixes`, `strict`, `empty_lines_in_values`, `default_section` and `interpolation` were added.

Modifié dans la version 3.5 : The `converters` argument was added.

defaults()

Return a dictionary containing the instance-wide defaults.

sections()

Return a list of the sections available; the *default section* is not included in the list.

add_section(section)

Add a section named *section* to the instance. If a section by the given name already exists, `DuplicateSectionError` is raised. If the *default section* name is passed, `ValueError` is raised. The name of the section must be a string; if not, `TypeError` is raised.

Modifié dans la version 3.2 : Non-string section names raise `TypeError`.

has_section(section)

Indicates whether the named *section* is present in the configuration. The *default section* is not acknowledged.

options(section)

Return a list of options available in the specified *section*.

has_option(section, option)

If the given *section* exists, and contains the given *option*, return `True`; otherwise return `False`. If the specified *section* is `None` or an empty string, `DEFAULT` is assumed.

read(filenames, encoding=None)

Attempt to read and parse an iterable of filenames, returning a list of filenames which were successfully parsed.

If *filenames* is a string or *path-like object*, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify an iterable of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the iterable will be read.

If none of the named files exist, the `ConfigParser` instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using `read_file()` before calling `read()` for any optional files :

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

Nouveau dans la version 3.2 : The *encoding* parameter. Previously, all files were read using the default encoding for `open()`.

Nouveau dans la version 3.6.1 : The *filenames* parameter accepts a *path-like object*.

read_file(f, source=None)

Read and parse configuration data from *f* which must be an iterable yielding Unicode strings (for example files opened in text mode).

Optional argument *source* specifies the name of the file being read. If not given and *f* has a `name` attribute, that is used for *source*; the default is '`<??>`'.

Nouveau dans la version 3.2 : Replaces `readfp()`.

read_string (*string*, *source*='<string>')

Parse configuration data from a string.

Optional argument *source* specifies a context-specific name of the string passed. If not given, '<string>' is used. This should commonly be a filesystem path or a URL.

Nouveau dans la version 3.2.

read_dict (*dictionary*, *source*='<dict>')

Load configuration from any object that provides a dict-like `items()` method. Keys are section names, values are dictionaries with keys and values that should be present in the section. If the used dictionary type preserves order, sections and their keys will be added in order. Values are automatically converted to strings. Optional argument *source* specifies a context-specific name of the dictionary passed. If not given, '<dict>' is used.

This method can be used to copy state between parsers.

Nouveau dans la version 3.2.

get (*section*, *option*, *, *raw*=False, *vars*=None[, *fallback*])

Get an *option* value for the named *section*. If *vars* is provided, it must be a dictionary. The *option* is looked up in *vars* (if provided), *section*, and in `DEFAULTSECT` in that order. If the key is not found and *fallback* is provided, it is used as a fallback value. `None` can be provided as a *fallback* value.

All the '%' interpolations are expanded in the return values, unless the *raw* argument is true. Values for interpolation keys are looked up in the same manner as the option.

Modifié dans la version 3.2 : Arguments *raw*, *vars* and *fallback* are keyword only to protect users from trying to use the third argument as the *fallback* fallback (especially when using the mapping protocol).

getint (*section*, *option*, *, *raw*=False, *vars*=None[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to an integer. See `get()` for explanation of *raw*, *vars* and *fallback*.

getfloat (*section*, *option*, *, *raw*=False, *vars*=None[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to a floating point number. See `get()` for explanation of *raw*, *vars* and *fallback*.

getboolean (*section*, *option*, *, *raw*=False, *vars*=None[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to a Boolean value. Note that the accepted values for the option are '1', 'yes', 'true', and 'on', which cause this method to return `True`, and '0', 'no', 'false', and 'off', which cause it to return `False`. These string values are checked in a case-insensitive manner. Any other value will cause it to raise `ValueError`. See `get()` for explanation of *raw*, *vars* and *fallback*.

items (*raw*=False, *vars*=None)

items (*section*, *raw*=False, *vars*=None)

When *section* is not given, return a list of *section_name*, *section_proxy* pairs, including `DEFAULTSECT`.

Otherwise, return a list of *name*, *value* pairs for the options in the given *section*. Optional arguments have the same meaning as for the `get()` method.

set (*section*, *option*, *value*)

If the given section exists, set the given option to the specified value; otherwise raise `NoSectionError`. *option* and *value* must be strings; if not, `TypeError` is raised.

write (*fileobject*, *space_around_delimiters*=True)

Write a representation of the configuration to the specified *file object*, which must be opened in text mode (accepting strings). This representation can be parsed by a future `read()` call. If *space_around_delimiters* is true, delimiters between keys and values are surrounded by spaces.

remove_option (*section*, *option*)

Remove the specified *option* from the specified *section*. If the section does not exist, raise `NoSectionError`. If the option existed to be removed, return `True`; otherwise return `False`.

remove_section (*section*)

Remove the specified *section* from the configuration. If the section in fact existed, return `True`. Otherwise return `False`.

optionxform (*option*)

Transforms the option name *option* as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of *option*; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don't need to subclass the parser to use this method, you can also set it on an instance, to a function that takes a string argument and returns a string. Setting it to `str`, for example, would make option names case sensitive :

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

Note that when reading configuration files, whitespace around the option names is stripped before `optionxform()` is called.

readfp (*fp*, *filename=None*)

Obsolète depuis la version 3.2 : Use `read_file()` instead.

Modifié dans la version 3.2 : `readfp()` now iterates on *fp* instead of calling `fp.readline()`.

For existing code calling `readfp()` with arguments which don't support iteration, the following generator may be used as a wrapper around the file-like object :

```
def readline_generator(fp):
    line = fp.readline()
    while line:
        yield line
        line = fp.readline()
```

Instead of `parser.readfp(fp)` use `parser.read_file(readline_generator(fp))`.

configparser.MAX_INTERPOLATION_DEPTH

The maximum depth for recursive interpolation for `get()` when the *raw* parameter is false. This is relevant only when the default *interpolation* is used.

14.2.10 RawConfigParser Objects

```
class configparser.RawConfigParser (defaults=None, dict_type=collections.OrderedDict,
                                     low_no_value=False, *, delimiters=('=', ':'),
                                     comment_prefixes=(';', '#'), inline_comment_prefixes=None,
                                     strict=True, empty_lines_in_values=True,
                                     default_section=configparser.DEFAULTSECT[, interpolation])
```

Legacy variant of the `ConfigParser` with interpolation disabled by default and unsafe `add_section` and `set` methods.

Note : Consider using `ConfigParser` instead which checks types of the values to be stored internally. If you don't want interpolation, you can use `ConfigParser(interpolation=None)`.

add_section (*section*)

Add a section named *section* to the instance. If a section by the given name already exists, `DuplicateSectionError` is raised. If the *default section* name is passed, `ValueError` is raised.

Type of *section* is not checked which lets users create non-string named sections. This behaviour is unsupported and may cause internal errors.

set (*section*, *option*, *value*)

If the given section exists, set the given option to the specified value; otherwise raise `NoSectionError`. While it is possible to use `RawConfigParser` (or `ConfigParser` with *raw* parameters set to true) for *internal* storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

This method lets users assign non-string values to keys internally. This behaviour is unsupported and will cause errors when attempting to write to a file or get it in non-raw mode. **Use the mapping protocol API** which does not allow such assignments to take place.

14.2.11 Exceptions

exception `configparser.Error`

Base class for all other *configparser* exceptions.

exception `configparser.NoSectionError`

Exception raised when a specified section is not found.

exception `configparser.DuplicateSectionError`

Exception raised if `add_section()` is called with the name of a section that is already present or in strict parsers when a section is found more than once in a single input file, string or dictionary.

Nouveau dans la version 3.2 : Optional `source` and `lineno` attributes and arguments to `__init__()` were added.

exception `configparser.DuplicateOptionError`

Exception raised by strict parsers if a single option appears twice during reading from a single file, string or dictionary. This catches misspellings and case sensitivity-related errors, e.g. a dictionary may have two keys representing the same case-insensitive configuration key.

exception `configparser.NoOptionError`

Exception raised when a specified option is not found in the specified section.

exception `configparser.InterpolationError`

Base class for exceptions raised when problems occur performing string interpolation.

exception `configparser.InterpolationDepthError`

Exception raised when string interpolation cannot be completed because the number of iterations exceeds `MAX_INTERPOLATION_DEPTH`. Subclass of *InterpolationError*.

exception `configparser.InterpolationMissingOptionError`

Exception raised when an option referenced from a value does not exist. Subclass of *InterpolationError*.

exception `configparser.InterpolationSyntaxError`

Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of *InterpolationError*.

exception `configparser.MissingSectionHeaderError`

Exception raised when attempting to parse a file which has no section headers.

exception `configparser.ParsingError`

Exception raised when errors occur attempting to parse a file.

Modifié dans la version 3.2 : The `filename` attribute and `__init__()` argument were renamed to `source` for consistency.

Notes

14.3 netrc — traitement de fichier *netrc*

Code source : [Lib/netrc.py](#)

La classe *netrc* analyse et encapsule le format de fichier *netrc* utilisé par le programme Unix **ftp** et d'autres clients FTP.

class `netrc.netrc([file])`

A *netrc* instance or subclass instance encapsulates data from a *netrc* file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file `.netrc` in the user's home directory will be read. Parse errors will raise *NetrcParseError* with diagnostic information including the file name, line number, and terminating token. If no argument is specified on a POSIX system, the presence of passwords in the `.netrc` file will raise a *NetrcParseError* if the file ownership or permissions are insecure (owned by a user other than the user running the process, or accessible for read or write by any other user). This implements security behavior equivalent to that of **ftp** and other programs that use `.netrc`.

Modifié dans la version 3.4 : Ajout de la vérification d'autorisations POSIX.

exception `netrc.NetrcParseError`

Exception levée par la classe *netrc* lorsque des erreurs syntaxiques sont rencontrées dans le texte source. Les instances de cette exception fournissent trois attributs intéressants : `msg` est une explication textuelle de l'erreur, `filename` est le nom du fichier source et `lineno` donne le numéro de la ligne sur laquelle l'erreur a été trouvée.

14.3.1 Objets *netrc*

Une instance *netrc* a les méthodes suivantes :

`netrc.authenticators(host)`

Renvoie un triplet (`login`, `account`, `password`) pour s'authentifier auprès de l'hôte *host*. Si le fichier *netrc* ne contient pas d'entrée pour l'hôte donné, renvoie le tuple associé à l'entrée par défaut. Si aucun hôte correspondant ni aucune entrée par défaut n'est disponible, renvoie `None`.

`netrc.__repr__()`

Déverse les données de la classe sous forme de chaîne dans le format d'un fichier *netrc*. (Ceci ignore les commentaires et peut réorganiser les entrées).

Les instances de *netrc* ont des variables d'instance publiques :

`netrc.hosts`

Dictionnaire faisant correspondre les noms d'hôtes dans des tuples (`login`, `account`, `password`). L'entrée par défaut, le cas échéant, est représentée en tant que pseudo-hôte par ce nom.

`netrc.macros`

Dictionnaire faisant correspondre les noms de macro en listes de chaînes.

Note : Les mots de passe sont limités à un sous-ensemble du jeu de caractères ASCII. Toute ponctuation ASCII est autorisée dans les mots de passe, cependant notez que les espaces et les caractères non imprimables ne sont pas autorisés dans les mots de passe. C'est une limitation de la façon dont le fichier *netrc* est analysé et pourra être supprimée à l'avenir.

14.4 `xdrlib` — Encode and decode XDR data

Code source : [Lib/xdrlib.py](#)

The `xdrlib` module supports the External Data Representation Standard as described in [RFC 1014](#), written by Sun Microsystems, Inc. June 1987. It supports most of the data types described in the RFC.

The `xdrlib` module defines two classes, one for packing variables into XDR representation, and another for unpacking from XDR representation. There are also two exception classes.

class `xdrlib.Packer`

Packer is the class for packing data into XDR representation. The *Packer* class is instantiated with no arguments.

class `xdrlib.Unpacker` (*data*)

Unpacker is the complementary class which unpacks XDR data values from a string buffer. The input buffer is given as *data*.

Voir aussi :

RFC 1014 - XDR : External Data Representation Standard This RFC defined the encoding of data which was XDR at the time this module was originally written. It has apparently been obsoleted by [RFC 1832](#).

RFC 1832 - XDR : External Data Representation Standard Newer RFC that provides a revised definition of XDR.

14.4.1 Packer Objects

Packer instances have the following methods :

`Packer.get_buffer()`

Returns the current pack buffer as a string.

`Packer.reset()`

Resets the pack buffer to the empty string.

In general, you can pack any of the most common XDR data types by calling the appropriate `pack_type()` method. Each method takes a single argument, the value to pack. The following simple data type packing methods are supported : `pack_uint()`, `pack_int()`, `pack_enum()`, `pack_bool()`, `pack_uhyper()`, and `pack_hyper()`.

`Packer.pack_float(value)`

Packs the single-precision floating point number *value*.

`Packer.pack_double(value)`

Packs the double-precision floating point number *value*.

The following methods support packing strings, bytes, and opaque data :

`Packer.pack_fstring(n, s)`

Packs a fixed length string, *s*. *n* is the length of the string but it is *not* packed into the data buffer. The string is padded with null bytes if necessary to guaranteed 4 byte alignment.

`Packer.pack_fopaque(n, data)`

Packs a fixed length opaque data stream, similarly to `pack_fstring()`.

`Packer.pack_string(s)`

Packs a variable length string, *s*. The length of the string is first packed as an unsigned integer, then the string data is packed with `pack_fstring()`.

`Packer.pack_opaque(data)`

Packs a variable length opaque data string, similarly to `pack_string()`.

`Packer.pack_bytes(bytes)`

Packs a variable length byte stream, similarly to `pack_string()`.

The following methods support packing arrays and lists :

`Packer.pack_list(list, pack_item)`

Packs a *list* of homogeneous items. This method is useful for lists with an indeterminate size ; i.e. the size is not available until the entire list has been walked. For each item in the list, an unsigned integer 1 is packed first, followed by the data value from the list. *pack_item* is the function that is called to pack the individual item. At the end of the list, an unsigned integer 0 is packed.

For example, to pack a list of integers, the code might appear like this :

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

`Packer.pack_farray(n, array, pack_item)`

Packs a fixed length list (*array*) of homogeneous items. *n* is the length of the list ; it is *not* packed into the buffer, but a `ValueError` exception is raised if `len(array)` is not equal to *n*. As above, *pack_item* is the function used to pack each element.

`Packer.pack_array(list, pack_item)`

Packs a variable length *list* of homogeneous items. First, the length of the list is packed as an unsigned integer, then each element is packed as in `pack_farray()` above.

14.4.2 Unpacker Objects

The `Unpacker` class offers the following methods :

`Unpacker.reset(data)`

Resets the string buffer with the given *data*.

`Unpacker.get_position()`

Returns the current unpack position in the data buffer.

`Unpacker.set_position(position)`

Sets the data buffer unpack position to *position*. You should be careful about using `get_position()` and `set_position()`.

`Unpacker.get_buffer()`

Returns the current unpack data buffer as a string.

`Unpacker.done()`

Indicates unpack completion. Raises an `Error` exception if all of the data has not been unpacked.

In addition, every data type that can be packed with a `Packer`, can be unpacked with an `Unpacker`. Unpacking methods are of the form `unpack_type()`, and take no arguments. They return the unpacked object.

`Unpacker.unpack_float()`

Unpacks a single-precision floating point number.

`Unpacker.unpack_double()`

Unpacks a double-precision floating point number, similarly to `unpack_float()`.

In addition, the following methods unpack strings, bytes, and opaque data :

`Unpacker.unpack_fstring(n)`

Unpacks and returns a fixed length string. *n* is the number of characters expected. Padding with null bytes to guaranteed 4 byte alignment is assumed.

`Unpacker.unpack_fopaque(n)`

Unpacks and returns a fixed length opaque data stream, similarly to `unpack_fstring()`.

`Unpacker.unpack_string()`

Unpacks and returns a variable length string. The length of the string is first unpacked as an unsigned integer, then the string data is unpacked with `unpack_fstring()`.

`Unpacker.unpack_opaque()`

Unpacks and returns a variable length opaque data string, similarly to `unpack_string()`.

`Unpacker.unpack_bytes()`

Unpacks and returns a variable length byte stream, similarly to `unpack_string()`.

The following methods support unpacking arrays and lists :

`Unpacker.unpack_list(unpack_item)`

Unpacks and returns a list of homogeneous items. The list is unpacked one element at a time by first unpacking an unsigned integer flag. If the flag is 1, then the item is unpacked and appended to the list. A flag of 0 indicates the end of the list. `unpack_item` is the function that is called to unpack the items.

`Unpacker.unpack_farray(n, unpack_item)`

Unpacks and returns (as a list) a fixed length array of homogeneous items. *n* is number of list elements to expect in the buffer. As above, `unpack_item` is the function used to unpack each element.

`Unpacker.unpack_array(unpack_item)`

Unpacks and returns a variable length *list* of homogeneous items. First, the length of the list is unpacked as an unsigned integer, then each element is unpacked as in `unpack_farray()` above.

14.4.3 Exceptions

Exceptions in this module are coded as class instances :

exception `xdrlib.Error`

The base exception class. `Error` has a single public attribute `msg` containing the description of the error.

exception `xdrlib.ConversionError`

Class derived from `Error`. Contains no additional instance variables.

Here is an example of how you would catch one of these exceptions :

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print('packing the double failed:', instance.msg)
```


14.5 plistlib — Generate and parse Mac OS X .plist files

Code source : [Lib/plistlib.py](#)

This module provides an interface for reading and writing the « property list » files used mainly by Mac OS X and supports both binary and XML plist files.

The property list (.plist) file format is a simple serialization supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

To write out and to parse a plist file, use the `dump()` and `load()` functions.

To work with plist data in bytes objects, use `dumps()` and `loads()`.

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), `Data`, `bytes`, `bytesarray` or `datetime.datetime` objects.

Modifié dans la version 3.4 : New API, old API deprecated. Support for binary format plists added.

Voir aussi :

PList manual page Apple's documentation of the file format.

Ce module définit les fonctions suivantes :

`plistlib.load(fp, *, fmt=None, use_builtin_types=True, dict_type=dict)`

Read a plist file. *fp* should be a readable and binary file object. Return the unpacked root object (which usually is a dictionary).

The *fmt* is the format of the file and the following values are valid :

- `None` : Autodetect the file format
- `FMT_XML` : XML file format
- `FMT_BINARY` : Binary plist format

If *use_builtin_types* is true (the default) binary data will be returned as instances of `bytes`, otherwise it is returned as instances of `Data`.

The *dict_type* is the type used for dictionaries that are read from the plist file. The exact structure of the plist can be recovered by using `collections.OrderedDict` (although the order of keys shouldn't be important in plist files).

XML data for the `FMT_XML` format is parsed using the Expat parser from `xml.parsers.expat` – see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

The parser for the binary format raises `InvalidFileException` when the file cannot be parsed.

Nouveau dans la version 3.4.

`plistlib.loads(data, *, fmt=None, use_builtin_types=True, dict_type=dict)`

Load a plist from a bytes object. See `load()` for an explanation of the keyword arguments.

Nouveau dans la version 3.4.

`plistlib.dump(value, fp, *, fmt=FMT_XML, sort_keys=True, skipkeys=False)`

Write *value* to a plist file. *Fp* should be a writable, binary file object.

The *fmt* argument specifies the format of the plist file and can be one of the following values :

- `FMT_XML` : XML formatted plist file
- `FMT_BINARY` : Binary formatted plist file

When *sort_keys* is true (the default) the keys for dictionaries will be written to the plist in sorted order, otherwise they will be written in the iteration order of the dictionary.

When *skipkeys* is false (the default) the function raises `TypeError` when a key of a dictionary is not a string, otherwise such keys are skipped.

A `TypeError` will be raised if the object is of an unsupported type or a container that contains objects of unsupported types.

An `OverflowError` will be raised for integer values that cannot be represented in (binary) plist files.

Nouveau dans la version 3.4.

`plistlib.dumps (value, *, fmt=FMT_XML, sort_keys=True, skipkeys=False)`

Return *value* as a plist-formatted bytes object. See the documentation for `dump()` for an explanation of the keyword arguments of this function.

Nouveau dans la version 3.4.

The following functions are deprecated :

`plistlib.readPlist (pathOrFile)`

Read a plist file. *pathOrFile* may be either a file name or a (readable and binary) file object. Returns the unpacked root object (which usually is a dictionary).

This function calls `load()` to do the actual work, see the documentation of *that function* for an explanation of the keyword arguments.

Note : Dict values in the result have a `__getattr__` method that defers to `__getitem__`. This means that you can use attribute access to access items of these dictionaries.

Obsolète depuis la version 3.4 : Use `load()` instead.

`plistlib.writePlist (rootObject, pathOrFile)`

Write *rootObject* to an XML plist file. *pathOrFile* may be either a file name or a (writable and binary) file object

Obsolète depuis la version 3.4 : Use `dump()` instead.

`plistlib.readPlistFromBytes (data)`

Read a plist data from a bytes object. Return the root object.

See `load()` for a description of the keyword arguments.

Note : Dict values in the result have a `__getattr__` method that defers to `__getitem__`. This means that you can use attribute access to access items of these dictionaries.

Obsolète depuis la version 3.4 : Use `loads()` instead.

`plistlib.writePlistToBytes (rootObject)`

Return *rootObject* as an XML plist-formatted bytes object.

Obsolète depuis la version 3.4 : Use `dumps()` instead.

The following classes are available :

Dict ([dict]) :

Return an extended mapping object with the same value as dictionary *dict*.

This class is a subclass of `dict` where attribute access can be used to access items. That is, `aDict.key` is the same as `aDict['key']` for getting, setting and deleting items in the mapping.

Obsolète depuis la version 3.0.

class plistlib.Data (data)

Return a « data » wrapper object around the bytes object *data*. This is used in functions converting from/to plists to represent the `<data>` type available in plists.

It has one attribute, `data`, that can be used to retrieve the Python bytes object stored in it.

Obsolète depuis la version 3.4 : Use a `bytes` object instead.

The following constants are available :

`plistlib.FMT_XML`

The XML format for plist files.

Nouveau dans la version 3.4.

`plistlib.FMT_BINARY`

The binary format for plist files

Nouveau dans la version 3.4.

14.5.1 Exemples

Generating a plist :

```
pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\ue4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime()))),
)
with open(fileName, 'wb') as fp:
    dump(pl, fp)
```

Parsing a plist :

```
with open(fileName, 'rb') as fp:
    pl = load(fp)
    print(pl["aKey"])
```

Service de cryptographie

Les modules décrits dans ce chapitre mettent en œuvre divers algorithmes cryptographiques. Ils peuvent, ou pas, être disponibles, en fonction de l'installation. Sur les systèmes Unix, le module `crypt` peut aussi être disponible. Voici une vue d'ensemble :

15.1 `hashlib` — Algorithmes de hachage sécurisés et synthèse de messages

Code source : [Lib/hashlib.py](#)

Ce module implémente une interface commune à différents algorithmes de hachage sécurisés et de synthèse de messages. Sont inclus les algorithmes standards FIPS de hachage SHA1, SHA224, SHA256, SHA384, et SHA512 (définis dans FIPS 180-2) ainsi que l'algorithme MD5 de RSA (défini par la [RFC 1321](#)). Les termes « algorithmes de hachage sécurisé » et « algorithme de synthèse de message » sont interchangeables. Les anciens algorithmes étaient appelés « algorithmes de synthèse de messages ». Le terme moderne est « algorithme de hachage sécurisé ».

Note : Si vous préférez utiliser les fonctions de hachage `adler32` ou `crc32`, elles sont disponibles dans le module `zlib`.

Avertissement : Certains algorithmes ont des faiblesses connues relatives à la collision, se référer à la section « Voir aussi » à la fin.

15.1.1 Algorithmes de hachage

Il y a un constructeur nommé selon chaque type de *hash*. Tous retournent un objet haché avec la même interface. Par exemple : utilisez `sha256()` pour créer un objet haché de type SHA-256. Vous pouvez maintenant utiliser cet objet *bytes-like objects* (normalement des *bytes*) en utilisant la méthode `update()`. À tout moment vous pouvez demander le *digest* de la concaténation des données fournies en utilisant les méthodes `digest()` ou `hexdigest()`.

Note : Pour de meilleures performances avec de multiples fils d'exécution, le *GIL* Python est relâché pour des données dont la taille est supérieure à 2047 octets lors de leur création ou leur mise à jour.

Note : Fournir des objets chaînes de caractères à la méthode `update()` n'est pas implémenté, comme les fonctions de hachages travaillent sur des *bytes* et pas sur des caractères.

Les constructeurs pour les algorithmes de hachage qui sont toujours présents dans ce module sont `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, `blake2b()`, et `blake2s()`. `md5()` est normalement disponible aussi, mais il peut être manquant si vous utilisez une forme rare de Python « conforme FIPS ». Des algorithmes additionnels peuvent aussi être disponibles dépendant de la librairie OpenSSL que Python utilise sur votre plate-forme. Sur la plupart des plates-formes les fonctions `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()` sont aussi disponibles.

Nouveau dans la version 3.6 : Les constructeurs SHA3 (Keccak) et SHAKE `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`.

Nouveau dans la version 3.6 : Les fonctions `blake2b()` et `blake2s()` ont été ajoutées.

Par exemple, pour obtenir l'empreinte de la chaîne `b'Nobody inspects the spammish repetition'`:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xddAe\x15\x93\xc5\xfe\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\
↪x0fK\x94\x06'
>>> m.digest_size
32
>>> m.block_size
64
```

En plus condensé :

```
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

`hashlib.new(name[, data])`

Is a generic constructor that takes the string *name* of the desired algorithm as its first parameter. It also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than `new()` and should be preferred.

En utilisant `new()` avec un algorithme fourni par OpenSSL :

```
>>> h = hashlib.new('ripemd160')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Hashlib fournit les constantes suivantes :

hashlib.algorithms_guaranteed

Un ensemble contenant les noms des algorithmes de hachage garantis d'être implémentés par ce module sur toutes les plate-formes. Notez que *md5* est dans cette liste malgré certains éditeurs qui offrent une implémentation Python de la librairie compatible FIPS l'excluant.

Nouveau dans la version 3.2.

hashlib.algorithms_available

Un ensemble contenant les noms des algorithmes de hachage disponibles dans l'interpréteur Python. Ces noms sont reconnus lorsqu'ils sont passés à la fonction *new()*. *algorithms_guaranteed* est toujours un sous-ensemble. Le même algorithme peut apparaître plusieurs fois dans cet ensemble sous un nom différent (grâce à OpenSSL).

Nouveau dans la version 3.2.

Les valeurs suivantes sont fournis en tant qu'attributs constants des objets hachés retournés par les constructeurs :

hash.digest_size

La taille du *hash* résultant en octets.

hash.block_size

La taille interne d'un bloc de l'algorithme de hachage en octets.

L'objet haché possède les attributs suivants :

hash.name

Le nom canonique de cet objet haché, toujours en minuscule et toujours transmissible à la fonction *new()* pour créer un autre objet haché de ce type.

Modifié dans la version 3.4 : L'attribut *name* est présent dans CPython depuis sa création, mais n'était pas spécifié formellement jusqu'à Python 3.4, il peut ne pas exister sur certaines plate-formes.

L'objet haché possède les méthodes suivantes :

hash.update (data)

Update the hash object with the *bytes-like object*. Repeated calls are equivalent to a single call with the concatenation of all the arguments : *m.update (a) ; m.update (b)* is equivalent to *m.update (a+b)*.

Modifié dans la version 3.1 : Le GIL Python est relâché pour permettre aux autres fils d'exécution de tourner pendant que la fonction de hachage met à jour des données plus larges que 2047 octets, lorsque les algorithmes fournis par OpenSSL sont utilisés.

hash.digest ()

Renvoie le *digest* des données passées à la méthode *update()*. C'est un objet de type *bytes* de taille *digest_size* qui contient des octets dans l'intervalle 0 à 255.

hash.hexdigest ()

Comme la méthode *digest()* sauf que le *digest* renvoyé est une chaîne de caractères de longueur double, contenant seulement des chiffres hexadécimaux. Cela peut être utilisé pour échanger sans risque des valeurs dans les *e-mails* ou dans les environnements non binaires.

hash.copy ()

Renvoie une copie (« clone ») de l'objet haché. Cela peut être utilisé pour calculer efficacement les *digests* de données partageant des sous-chaînes communes.

15.1.2 Synthèse de messages de taille variable SHAKE

Les algorithmes `shake_128()` et `shake_256()` fournissent des messages de longueur variable avec des longueurs en bits // 2 jusqu'à 128 ou 256 bits de sécurité. Leurs méthodes *digests* requièrent une longueur. Les longueurs maximales ne sont pas limitées par l'algorithme SHAKE.

`shake.digest (length)`

Return the digest of the data passed to the `update()` method so far. This is a bytes object of size *length* which may contain bytes in the whole range from 0 to 255.

`shake.hexdigest (length)`

Comme la méthode *digest()* sauf que le *digest* renvoyé est une chaîne de caractères de longueur double, contenant seulement des chiffres hexadécimaux. Cela peut être utilisé pour échanger sans risque des valeurs dans les *e-mails* ou dans les environnements non binaires.

15.1.3 Dérivation de clé

Les algorithmes de dérivation de clés et d'étirement de clés sont conçus pour le hachage sécurisé de mots de passe. Des algorithmes naïfs comme `sha1(password)` ne sont pas résistants aux attaques par force brute. Une bonne fonction de hachage doit être paramétrable, lente, et inclure un [sel](#).

`hashlib.pbkdf2_hmac (hash_name, password, salt, iterations, dklen=None)`

La fonction fournit une fonction de dérivation PKCS#5 (*Public Key Cryptographic Standards #5 v2.0*). Elle utilise HMAC comme fonction de pseudo-aléatoire.

La chaîne de caractères *hash_name* est le nom de l'algorithme de hachage désiré pour le HMAC, par exemple "sha1" ou "sha256". *password* et *salt* sont interprétés comme des tampons d'octets. Les applications et bibliothèques doivent limiter *password* à une longueur raisonnable (comme 1024). *salt* doit être de 16 octets ou plus provenant d'une source correcte, e.g. `os.urandom()`.

Le nombre d'*iterations* doit être choisi sur la base de l'algorithme de hachage et de la puissance de calcul. En 2013, au moins 100000 itérations de SHA-256 sont recommandées.

dklen est la longueur de la clé dérivée. Si *dklen* vaut None alors la taille du message de l'algorithme de hachage *hash_name* est utilisé, e.g. 64 pour SHA-512.

```
>>> import hashlib, binascii
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> binascii.hexlify(dk)
b'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

Nouveau dans la version 3.4.

Note : Une implémentation rapide de *pbkdf2_hmac* est disponible avec OpenSSL. L'implémentation Python utilise une version anonyme de *hmac*. Elle est trois fois plus lente et ne libère pas le GIL.

`hashlib.scrypt (password, *, salt, n, r, p, maxmem=0, dklen=64)`

La fonction fournit la fonction de dérivation de clé *scrypt* comme définie dans [RFC 7914](#).

password and *salt* must be *bytes-like objects*. Applications and libraries should limit *password* to a sensible length (e.g. 1024). *salt* should be about 16 or more bytes from a proper source, e.g. `os.urandom()`.

n is the CPU/Memory cost factor, *r* the block size, *p* parallelization factor and *maxmem* limits memory (OpenSSL 1.1.0 defaults to 32 MB). *dklen* is the length of the derived key.

Availability : OpenSSL 1.1+

Nouveau dans la version 3.6.

15.1.4 BLAKE2

BLAKE2 est une fonction de hachage cryptographique définie dans la **RFC 7693** et disponible en deux versions :

- **BLAKE2b**, optimisée pour les plates-formes 64-bit et produisant des messages de toutes tailles entre 1 et 64 octets,
- **BLAKE2s**, optimisée pour les plates-formes de 8 à 32-bit et produisant des messages de toutes tailles entre 1 et 32 octets.

BLAKE2 gère diverses fonctionnalités **keyed mode** (un remplacement plus rapide et plus simple pour **HMAC**), **salted hashing**, **personalization**, et **tree hashing**.

Les objets hachés de ce module suivent l'API des objets du module `hashlib` de la librairie standard.

Création d'objets hachés

Les nouveaux objets hachés sont créés en appelant les constructeurs :

```
hashlib.blake2b(data=b", *, digest_size=64, key=b", salt=b", person=b", fanout=1, depth=1, leaf_size=0,
                node_offset=0, node_depth=0, inner_size=0, last_node=False)
```

```
hashlib.blake2s(data=b", *, digest_size=32, key=b", salt=b", person=b", fanout=1, depth=1, leaf_size=0,
                node_offset=0, node_depth=0, inner_size=0, last_node=False)
```

Ces fonctions produisent l'objet haché correspondant aux calculs de BLAKE2b ou BLAKE2s. Elles prennent ces paramètres optionnels :

- *data* : initial chunk of data to hash, which must be *bytes-like object*. It can be passed only as positional argument.
- *digest_size* : taille en sortie du message en octets.
- *key* : clé pour les code d'authentification de message *keyed hashing* (jusqu'à 64 octets pour BLAKE2b, jusqu'à 32 octets pour BLAKE2s).
- *salt* : sel pour le hachage randomisé *randomized hashing* (jusqu'à 16 octets pour BLAKE2b, jusqu'à 8 octets pour BLAKE2s).
- *person* : chaîne de personnalisation (jusqu'à 16 octets pour BLAKE2b, jusqu'à 8 octets pour BLAKE2s).

Le tableau suivant présente les limites des paramètres généraux (en octets) :

Hash	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

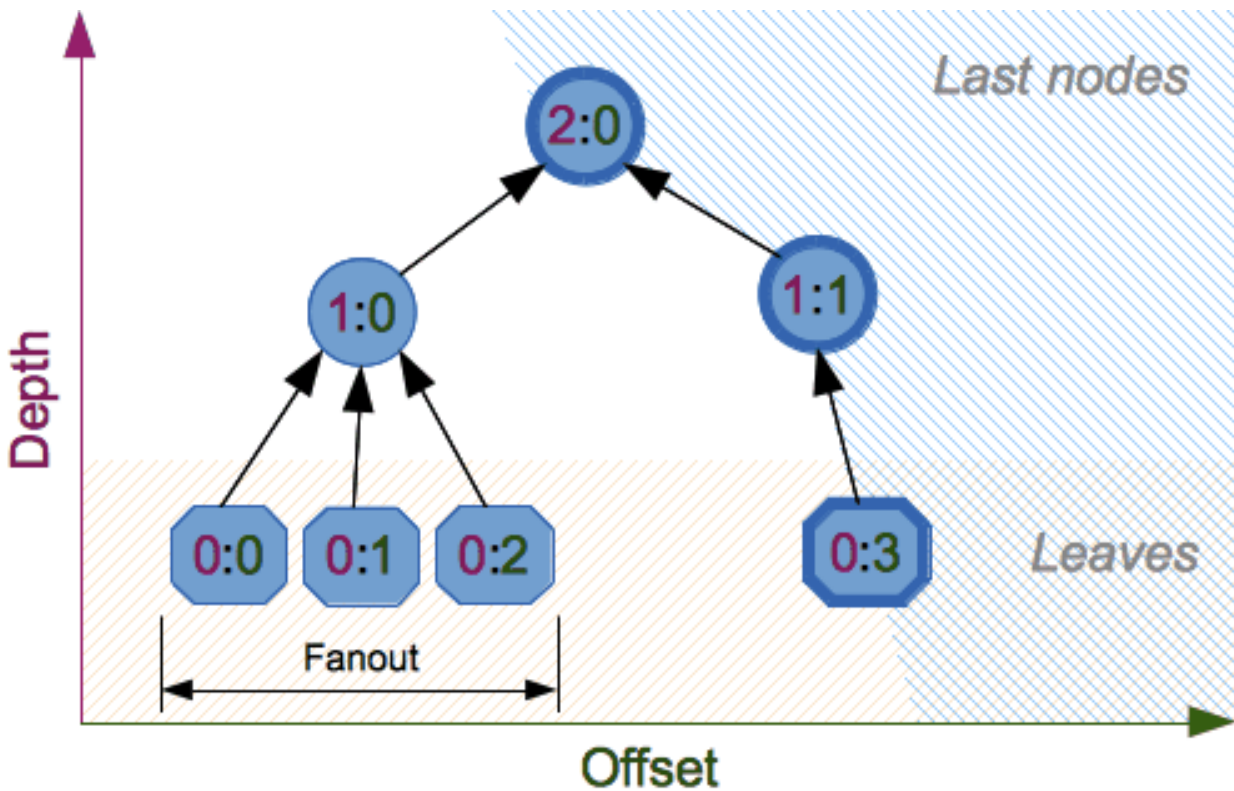
Note : Les spécifications de BLAKE2 définissent des longueurs constantes pour les sel et chaînes de personnalisation, toutefois, par commodité, cette implémentation accepte des chaînes *byte* de n'importe quelle taille jusqu'à la longueur spécifiée. Si la longueur du paramètre est moindre par rapport à celle spécifiée, il est complété par des zéros, ainsi, par exemple, `b'salt'` et `b'salt\x00'` sont la même valeur (Ce n'est pas le cas pour *key*.)

Ces tailles sont disponibles comme *constants* du module et décrites ci-dessous.

Les fonctions constructeur acceptent aussi les paramètres suivants pour le *tree hashing* :

- *fanout* : *fanout* (0 à 255, 0 si illimité, 1 en mode séquentiel).
- *depth* : profondeur maximale de l'arbre (1 à 255, 255 si illimité, 1 en mode séquentiel).
- *leaf_size* : taille maximale en octets d'une feuille (0 à $2^{**}32-1$, 0 si illimité ou en mode séquentiel).
- *node_offset* : décalage de nœud (0 à $2^{**}64-1$ pour BLAKE2b, 0 à $2^{**}48-1$ pour BLAKE2s, 0 pour la première feuille la plus à gauche, ou en mode séquentiel).
- *node_depth* : profondeur de nœuds (0 à 255, 0 pour les feuilles, ou en mode séquentiel).
- *inner_size* : taille interne du message (0 à 64 pour BLAKE2b, 0 à 32 pour BLAKE2s, 0 en mode séquentiel).
- *last_node* : booléen indiquant si le nœud traité est le dernier (*False* pour le mode séquentiel).

Voir section 2.10 dans **BLAKE2 specification** pour une approche compréhensive du *tree hashing*.



Constantes

`blake2b.SALT_SIZE`

`blake2s.SALT_SIZE`

Longueur du sel (longueur maximale acceptée par les constructeurs).

`blake2b.PERSON_SIZE`

`blake2s.PERSON_SIZE`

Longueur de la chaîne de personnalisation (longueur maximale acceptée par les constructeurs).

`blake2b.MAX_KEY_SIZE`

`blake2s.MAX_KEY_SIZE`

Taille maximale de clé.

`blake2b.MAX_DIGEST_SIZE`

`blake2s.MAX_DIGEST_SIZE`

Taille maximale du message que peut fournir la fonction de hachage.

Exemples

Hachage simple

Pour calculer les *hash* de certaines données, vous devez d'abord construire un objet haché en appelant la fonction constructeur appropriée (*blake2b()* or *blake2s()*), ensuite le mettre à jour avec les données en appelant la méthode *update()* sur l'objet, et, pour finir, récupérer l'empreinte du message en appelant la méthode *digest()* (ou *hexdigest()* pour les chaînes hexadécimales).

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪ '6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495d'
↪ ''
```

As a shortcut, you can pass the first chunk of data to update directly to the constructor as the positional argument :

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪ '6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495d'
↪ ''
```

Vous pouvez appeler la méthode *hash.update()* autant de fois que nécessaire pour mettre à jour le *hash* de manière itérative :

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
>>> h.hexdigest()

↪ '6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495d'
↪ ''
```

Usage de tailles d'empreintes différentes

BLAKE2 permet de configurer la taille des empreintes jusqu'à 64 octets pour BLAKE2b et jusqu'à 32 octets pour BLAKE2s. Par exemple, pour remplacer SHA-1 par BLAKE2b sans changer la taille de la sortie, nous pouvons dire à BLAKE2b de produire une empreinte de 20 octets :

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

Les objets hachés avec différentes tailles d'empreintes ont des sorties complètement différentes (les *hash* plus courts *ne sont pas* des préfixes de *hash* plus longs); BLAKE2b et BLAKE2s produisent des sorties différentes même si les longueurs des sorties sont les mêmes :

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

Code d'authentification de message

Keyed hashing can be used for authentication as a faster and simpler replacement for [Hash-based message authentication code](#) (HMAC). BLAKE2 can be securely used in prefix-MAC mode thanks to the indistinguishability property inherited from BLAKE.

Cet exemple montre comment obtenir un code d'authentification de message de 128-bit (en hexadécimal) pour un message `b'message data'` avec la clé `b'pseudorandom key'` :

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

Comme exemple pratique, une application web peut chiffrer symétriquement les *cookies* envoyés aux utilisateurs et les vérifier plus tard pour être certaine qu'ils n'aient pas été altérés :

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

Même s'il possède en natif la création de code d'authentification de message (MAC), BLAKE2 peut, bien sûr, être utilisé pour construire un HMAC en combinaison du module `hmac` :

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

Hachage randomisé

En définissant le paramètre *salt* les utilisateurs peuvent introduire de l'aléatoire dans la fonction de hachage. Le hachage randomisé est utile pour se protéger des attaques par collisions sur les fonctions de hachage utilisées dans les signatures numériques.

Randomized hashing is designed for situations where one party, the message preparer, generates all or part of a message to be signed by a second party, the message signer. If the message preparer is able to find cryptographic hash function collisions (i.e., two messages producing the same hash value), then they might prepare meaningful versions of the message that would produce the same hash value and digital signature, but with different results (e.g., transferring \$1,000,000 to an account, rather than \$10). Cryptographic hash functions have been designed with collision resistance as a major goal, but the current concentration on attacking cryptographic hash functions may result in a given cryptographic hash function providing less collision resistance than expected. Randomized hashing offers the signer additional protection by reducing the likelihood that a preparer can generate two or more messages that ultimately yield the same hash value during the digital signature generation process — even if it is practical to find collisions for the hash function. However, the use of randomized hashing may reduce the amount of security provided by a digital signature when all portions of the message are prepared by the signer.

(NIST SP-800-106 « Randomized Hashing for Digital Signatures »)

Dans BLAKE2, le sel est passé une seule fois lors de l'initialisation de la fonction de hachage, plutôt qu'à chaque appel d'une fonction de compression.

Avertissement : *Salted hashing* (ou juste hachage) avec BLAKE2 ou toute autre fonction de hachage générique, comme SHA-256, ne convient pas pour le chiffrement des mots de passe. Voir [BLAKE2 FAQ](#) pour plus d'informations.

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

Personnalisation

Parfois il est utile de forcer une fonction de hachage à produire différentes empreintes de message d'une même entrée pour différentes utilisations. Pour citer les auteurs de la fonction de hachage Skein :

Nous recommandons que tous les développeurs d'application considèrent sérieusement de faire cela ; nous avons vu de nombreux protocoles où un *hash* était calculé à un endroit du protocole pour être utilisé à un autre endroit car deux calculs de *hash* étaient réalisés sur des données similaires ou liées, et qu'un attaquant peut forcer une application à prendre en entrée le même *hash*. Personnaliser chaque fonction de hachage utilisée dans le protocole stoppe immédiatement ce genre d'attaque.

(The Skein Hash Function Family, p. 21, article en anglais)

BLAKE2 peut être personnalisé en passant des *bytes* à l'argument *person* :

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

La personnalisation et le *keyed mode* peuvent être utilisés ensemble pour dériver différentes clés à partir d'une seule.

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPLYk1e/nWfu0WSEb0KRcjhDeP/o=
```

Mode Arbre

L'exemple ci-dessous présente comment hacher un arbre minimal avec deux nœuds terminaux :

```
  10
 /  \
00  01
```

Cet exemple utilise en interne des empreintes de 64 octets, et produit finalement des empreintes 32 octets :

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
```

(suite sur la page suivante)

(suite de la page précédente)

```

>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...             leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...             node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...             leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...             node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...             leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...             node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'

```

Crédits

BLAKE2 a été conçu par *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn*, et *Christian Winnerlein* basé sur **SHA-3** finaliste **BLAKE** créé par *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier*, et *Raphael C.-W. Phan*.

Il utilise le cœur de l'algorithme de chiffrement de **ChaCha** conçu par *Daniel J. Bernstein*.

L'implémentation dans la librairie standard est basée sur le module **pyblake2**. Il a été écrit par *Dmitry Chestnykh* et basé sur l'implémentation C écrite par *Samuel Neves*. La documentation a été copiée depuis **pyblake2** et écrite par *Dmitry Chestnykh*.

Le code C a été partiellement réécrit pour Python par *Christian Heimes*.

Le transfert dans le domaine public s'applique pour l'implémentation C de la fonction de hachage, ses extensions et cette documentation :

Tout en restant dans les limites de la loi, le(s) auteur(s) a (ont) consacré tous les droits d'auteur et droits connexes et voisins de ce logiciel au domaine public dans le monde entier. Ce logiciel est distribué sans aucune garantie.

You should have received a copy of the CC0 Public Domain Dedication along with this software. If not, see <http://creativecommons.org/publicdomain/zero/1.0/>.

Les personnes suivantes ont aidé au développement ou contribué aux modification du projet et au domaine public selon la licence Creative Commons Public Domain Dedication 1.0 Universal :

— *Alexandr Sokolovskiy*

Voir aussi :

Module *hmac* Un module pour générer des codes d'authentification utilisant des *hash*.

Module *base64* Un autre moyen d'encoder des *hash* binaires dans des environnements non binaires.

<https://blake2.net> Site officiel de BLAKE2.

<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf> La publication FIPS 180-2 sur les algorithmes de hachage sécurisés.

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms Article Wikipedia contenant les informations relatives aux algorithmes ayant des problèmes et leur interprétation au regard de leur utilisation.

<https://www.ietf.org/rfc/rfc2898.txt> PKCS #5 : Password-Based Cryptography Specification Version 2.0

15.2 hmac — Authentification de messages par hachage en combinaison avec une clé secrète

Code source : [Lib/hmac.py](#)

Ce module implémente l'algorithme HMAC tel que décrit par la [RFC 2104](#).

`hmac.new(key, msg=None, digestmod=None)`

Return a new hmac object. *key* is a bytes or bytearray object giving the secret key. If *msg* is present, the method call `update(msg)` is made. *digestmod* is the digest name, digest constructor or module for the HMAC object to use. It supports any name suitable to `hashlib.new()` and defaults to the `hashlib.md5` constructor.

Modifié dans la version 3.4 : Le paramètre *key* peut être un *byte* ou un objet *bytearray*. Le paramètre *msg* peut être de n'importe quel type pris en charge par `hashlib`. Le paramètre *digestmod* peut être le nom d'un algorithme de hachage.

Obsolète depuis la version 3.4 : MD5 as implicit default digest for *digestmod* is deprecated.

Un objet HMAC a les méthodes suivantes :

`HMAC.update(msg)`

Met à jour l'objet HMAC avec *msg*. Des appels répétés sont équivalents à un seul appel avec la concaténation de tous les arguments : `m.update(a)` ; `m.update(b)` est équivalent à `m.update(a + b)`.

Modifié dans la version 3.4 : Le paramètre *msg* peut être de n'importe quel type géré par `hashlib`.

`HMAC.digest()`

Renvoie le condensat des octets passés à la méthode `update()` jusque là. L'objet *bytes* renvoyé sera de la même longueur que la *digest_size* de la fonction de hachage donnée au constructeur. Il peut contenir des octets qui ne sont pas dans la table ASCII, y compris des octets NUL.

Avertissement : Si vous devez vérifier la sortie de `digest()` avec un condensat obtenu par ailleurs par un service extérieur durant une routine de vérification, il est recommandé d'utiliser la fonction `compare_digest()` au lieu de l'opérateur `==` afin de réduire la vulnérabilité aux attaques temporelles.

`HMAC.hexdigest()`

Comme `digest()` sauf que ce condensat est renvoyé en tant que chaîne de caractères de taille doublée contenant seulement des chiffres hexadécimaux. Cela permet d'échanger le résultat sans problèmes par e-mail ou dans d'autres environnements ne gérant pas les données binaires.

Avertissement : Si l'on compare la sortie de `hexdigest()` avec celle d'un condensat connu obtenu par un service extérieur durant une routine de vérification, il est recommandé d'utiliser la fonction `compare_digest()` au lieu de l'opérateur `==` afin de réduire la vulnérabilité aux attaques basées sur les temps de réponse.

`HMAC.copy()`

Renvoie une copie (un clone) de l'objet HMAC. C'est utile pour calculer de manière efficace les empreintes cryptographiques de chaînes de caractères qui ont en commun une sous-chaîne initiale.

Un objet *code d'authentification de message* (HMAC) possède les attributs suivants :

`HMAC.digest_size`

La taille du code d'authentification (c-à-d de l'empreinte cryptographique) en octets.

`HMAC.block_size`

La taille interne d'un bloc de l'algorithme de hachage en octets.

Nouveau dans la version 3.4.

HMAC.name

Le nom canonique de ce HMAC, toujours en lettres minuscules, par exemple `hmac-md5`.

Nouveau dans la version 3.4.

Ce module fournit également la fonction utilitaire suivante :

hmac.compare_digest (*a*, *b*)

Renvoie `a == b`. Cette fonction a été conçue pour prévenir les attaques temporelles en évitant l'implémentation de courts-circuits basés sur le contenu, ce qui la rend appropriée pour de la cryptographie. *a* et *b* doivent être du même type : soit *str* (caractères ASCII seulement, comme retourné par `HMAC.hexdigest()` par exemple), ou *bytes-like object*.

Note : Si *a* et *b* sont de longueurs différentes ou si une erreur se produit, une attaque temporelle pourrait en théorie obtenir des informations sur les types et longueurs de *a* et de *b*, mais pas sur leurs valeurs.

Nouveau dans la version 3.3.

Voir aussi :

Module `hashlib` Le module Python fournissant des fonctions de hachage sécurisé.

15.3 secrets — Générer des nombres aléatoires de façon sécurisée pour la gestion des secrets

Nouveau dans la version 3.6.

Code source : [Lib/secrets.py](#)

Le module `secrets` permet de générer des nombres aléatoires forts au sens de la cryptographie, adaptés à la gestion des mots de passe, à l'authentification des comptes, à la gestion des jetons de sécurité et des secrets associés.

Il faut préférer `secrets` par rapport au générateur pseudo-aléatoire du module `random`, ce dernier étant conçu pour la modélisation et la simulation, et non pour la sécurité ou la cryptographie.

Voir aussi :

PEP 506

15.3.1 Nombres aléatoires

Le module `secrets` fournit un accès à la source d'aléa la plus sûre disponible sur votre système d'exploitation.

class secrets.SystemRandom

Classe permettant de générer des nombres aléatoires à partir des sources d'aléa les plus sûres fournies par le système d'exploitation. Se référer à `random.SystemRandom` pour plus de détails.

secrets.choice (*sequence*)

Renvoie un élément choisi aléatoirement dans une séquence non-vide.

secrets.randbelow (*n*)

Renvoie un entier aléatoire dans l'intervalle $[0, n)$.

secrets.randbits (*k*)

Renvoie un entier de *k* bits aléatoires.

15.3.2 Génération de jetons

Le module `secrets` fournit des fonctions pour la génération sécurisée de jetons adaptés à la réinitialisation de mots de passe, à la production d'URLs difficiles à deviner, etc.

`secrets.token_bytes([nbytes=None])`

Renvoie une chaîne d'octets aléatoire contenant *nbytes* octets. Si *nbytes* est `None` ou omis, une valeur par défaut raisonnable est utilisée.

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

`secrets.token_hex([nbytes=None])`

Renvoie une chaîne de caractères aléatoire en hexadécimal. La chaîne comporte *nbytes* octets aléatoires, chaque octet étant écrit sous la forme de deux chiffres hexadécimaux. Si *nbytes* est `None` ou omis, une valeur par défaut raisonnable est utilisée.

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

`secrets.token_urlsafe([nbytes=None])`

Renvoie une chaîne de caractères aléatoire adaptée au format URL, contenant *nbytes* octets aléatoires. Le texte est encodé en base64, chaque octet produisant en moyenne 1,3 caractères. Si *nbytes* est `None` ou omis, une valeur par défaut raisonnable est utilisée.

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

Combien d'octets mon jeton doit-il comporter ?

Afin de se prémunir des [attaques par force brute](#), les jetons doivent être suffisamment aléatoires. Malheureusement, l'augmentation de la puissance de calcul des ordinateurs leur permet de réaliser plus de tentatives dans le même laps de temps. De ce fait, le nombre de bits recommandé pour l'aléa augmente aussi. En 2015, une longueur de 32 octets (256 bits) aléatoires est généralement considérée suffisante pour les usages typiques du module `secrets`.

Si vous souhaitez gérer la longueur des jetons par vous-même, vous pouvez spécifier la quantité d'aléa à introduire dans les jetons en passant un argument *int* aux différentes fonctions `token_*`. Cet argument indique alors le nombre d'octets aléatoires utilisés pour la création du jeton.

Sinon, si aucun argument n'est passé ou si celui-ci est `None`, les fonctions `token_*` utilisent une valeur par défaut raisonnable à la place.

Note : Cette valeur par défaut est susceptible de changer à n'importe quel moment, y compris lors des mises à jour de maintenance.

15.3.3 Autres fonctions

`secrets.compare_digest(a, b)`

Return True if strings *a* and *b* are equal, otherwise False, in such a way as to reduce the risk of [timing attacks](#). See `hmac.compare_digest()` for additional details.

15.3.4 Recettes et bonnes pratiques

Cette section expose les recettes et les bonnes pratiques d'utilisation de `secrets` pour gérer un niveau minimal de sécurité.

Générer un mot de passe à huit caractères alphanumériques :

```
import string
alphabet = string.ascii_letters + string.digits
password = ''.join(choice(alphabet) for i in range(8))
```

Note : Les applications ne doivent jamais stocker des mots de passe dans un format permettant leur récupération, que ce soit en texte brut ou chiffré. Il convient que les mots de passe soient salés et transformés de façon irréversible par une fonction de hachage cryptographique.

Générer un mot de passe alphanumérique à dix caractères contenant au moins un caractère en minuscule, au moins un caractère en majuscule et au moins trois chiffres :

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
        and any(c.isupper() for c in password)
        and sum(c.isdigit() for c in password) >= 3):
        break
```

Generate an XKCD-style passphrase :

```
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(choice(words) for i in range(4))
```

Générer une URL temporaire difficile à deviner contenant un jeton de sécurité adapté à réinitialisation d'un mot de passe :

```
url = 'https://mydomain.com/reset=' + token_urlsafe()
```

Services génériques du système d'exploitation

Les modules documentés dans ce chapitre fournissent des interfaces vers des fonctionnalités communes à la grande majorité des systèmes d'exploitation, telles que les fichiers et l'horloge. Bien que ces interfaces soient classiquement calquées sur les interfaces Unix ou C, elles sont aussi disponibles sur la plupart des autres systèmes. En voici un aperçu.

16.1 `os` — Diverses interfaces pour le système d'exploitation

Code source : [Lib/os.py](#)

Ce module fournit une manière portable d'utiliser les fonctionnalités dépendantes du système d'exploitation. Si vous voulez uniquement lire ou écrire dans un fichier, voir `open()`, si vous voulez manipuler des chemins, voir le module `os.path`, et si vous voulez lire toutes les lignes de tous les fichiers listés par l'invite de commande, voir le module `fileinput`. Pour créer des fichiers temporaires, voir le module `tempfile`, et pour une manipulation haut niveau des fichiers et dossiers, voir le module `shutil`.

Notes sur la disponibilité de ces fonctions :

- La conception des modules natifs Python dépendants du système d'exploitation est qu'une même fonctionnalité utilise une même interface. Par exemple, la fonction `os.stat(path)` renvoie des informations sur les statistiques de `path` dans le même format (qui est originaire de l'interface POSIX).
- Les extensions propres à un certain système d'exploitation sont également disponible par le module `os`, mais les utiliser est bien entendu une menace pour la portabilité.
- Toutes les fonctions acceptant les chemins ou noms de fichiers acceptent aussi bien des *bytes* que des *string*, et si un chemin ou un nom de fichier est renvoyé, il sera du même type.
- Une note « Disponibilité : Unix » signifie que cette fonction est communément implémentée dans les systèmes Unix. Une telle note ne prétend pas l'existence de la fonction sur un système d'exploitation particulier.
- Si ce n'est pas mentionné séparément, toutes les fonctions se réclamant « Disponibilité : Unix » sont gérées sur Mac OS X, qui est basé sur Unix.

Note : Toutes les fonctions de ce module lèvent une `OSError` dans le cas d'un chemin ou nom de fichier invalide ou inaccessible, ou si d'autres arguments sont de type correct mais non géré par le système d'exploitation.

exception `os.error`

Un alias pour les exceptions natives `OSError`.

`os.name`

Le nom du module importé dépendant du système d'exploitation. Les noms suivants ont actuellement été enregistrés : 'posix', 'nt', 'java'.

Voir aussi :

`sys.platform` a une granularité plus fine. `os.uname()` donne des informations de version dépendantes de système.

Le module `platform` fournit des vérifications détaillées pour l'identité du système.

16.1.1 Noms de fichiers, arguments en ligne de commande, et variables d'environnement

En Python, les noms de fichiers, les arguments en ligne de commandes, et les variables d'environnement sont représentées en utilisant le type *string*. Sur certains systèmes, décoder ces chaînes de caractères depuis et vers des *bytes* est nécessaire avant de les passer au système d'exploitation. Python utilise l'encodage du système de fichiers pour réaliser ces conversions (voir `sys.getfilesystemencoding()`).

Modifié dans la version 3.1 : Sur certains systèmes, les conversions utilisant l'encodage du système de fichiers peut échouer. Dans ce cas, Python utilise le *surrogateescape encoding error handler* (le gestionnaire d'erreurs d'encodage *surrogateescape*), ce qui veut dire que les bytes indécodables sont remplacés par un caractère Unicode U+DCxx au décodage, et ceux-ci sont retraduits en le bon octet à l'encodage.

L'encodage du système de fichiers doit garantir de pouvoir décoder correctement tous les octets en dessous de 128. Si l'encodage du système de fichiers ne peut garantir cela, les fonctions de l'API peuvent lever une `UnicodeError`.

16.1.2 Paramètres de processus

Ces fonctions et valeurs fournissent des informations et agissent sur le processus actuel et sur l'utilisateur.

`os.ctermid()`

Renvoie l'identifiant de fichier correspondant au terminal contrôlant le processus.

Disponibilité : Unix.

`os.environ`

Un objet *mapping* représentant les variables d'environnement. Par exemple `environ['HOME']` est le chemin vers votre répertoire d'accueil (sur certaines plate-formes), et est équivalent à `getenv("HOME")` en C.

Ce *mapping* est capturé la première fois que le module `os` est importé, typiquement pendant le démarrage de Python, lors de l'exécution de `site.py`. Les changements de l'environnement opérés après cette capture ne sont pas répercutés dans `os.environ`, à part les modifications directes de `os.environ`.

Si la plate-forme prend en charge la fonction `putenv()`, ce *mapping* peut être utilisé pour modifier l'environnement autant que pour l'interroger. `putenv()` sera appelée automatiquement quand le *mapping* sera modifié.

Sur Unix, les clefs et les valeurs utilisent `sys.getfilesystemencoding()` et le gestionnaire d'erreurs *surrogateescape*. Utilisez `environb` si vous désirez utiliser un encodage différent.

Note : Appeler `putenv()` ne change pas directement `os.environ`, donc il est préférable de modifier `os.environ`.

Note : Sur certaines plate-formes, dont FreeBSD et Mac OS X, procéder à des assignations sur `environ` peut causer des fuites de mémoire. Reférez-vous à la documentation système de `putenv()`.

Si `putenv()` n'est pas fourni, une copie modifiée de ce dictionnaire peut être passée aux fonctions appropriées de création de processus pour forcer l'utilisation d'un environnement modifié pour le processus fils.

Si la plate-forme prend en charge la fonction `unsetenv()`, vous pouvez supprimer des éléments de ce dictionnaire pour supprimer des variables d'environnement. La fonction `unsetenv()` sera appelée automatiquement quand un élément est supprimé de `os.environ`, ou quand l'une des méthodes `pop()` ou `clear()` est appelée.

`os.environb`

Une version en *bytes* de `environ` : un *mapping* d'objets représentant l'environnement par des chaîne de *bytes*. `environ` et `environb` sont synchronisés (modifier `environ` modifie `environb`, et vice-versa).

`environb` n'est disponible que si `supports_byte_environ` vaut `True`.

Nouveau dans la version 3.2.

`os.chdir(path)`

`os.fchdir(fd)`

`os.getcwd()`

Ces fonctions sont décrites dans *Fichiers et répertoires*.

`os.fsencode(filename)`

Encode le *chemin-compatible* `filename` vers l'encodage du système de fichiers avec une gestion d'erreurs `'surrogateescape'`, ou `'strict'` sous Windows; renvoie un objet *bytes* inchangé.

`fsdecode()` est la fonction inverse.

Nouveau dans la version 3.2.

Modifié dans la version 3.6 : Ajout de la prise en charge des objets implémentant l'interface `os.PathLike`.

`os.fsdecode(filename)`

Encode le *chemin-compatible* `filename` depuis l'encodage du système de fichiers avec une gestion d'erreurs `'surrogateescape'`, ou `'strict'` sous Windows; renvoie un objet *str* inchangé.

`fsencode()` est la fonction inverse.

Nouveau dans la version 3.2.

Modifié dans la version 3.6 : Ajout de la prise en charge des objets implémentant l'interface `os.PathLike`.

`os.fspath(path)`

Renvoie la représentation par le système de fichiers du chemin.

Si un objet *str* ou *bytes* est passé, il est renvoyé inchangé. Autrement, `__fspath__()` est appelée et sa valeur renvoyée tant qu'elle est un objet *str* ou *bytes*. Dans tous les autres cas, une `TypeError` est levée.

Nouveau dans la version 3.6.

`class os.PathLike`

Une *abstract base class* pour les objets représentant un chemin du système de fichiers, comme `pathlib.PurePath`.

Nouveau dans la version 3.6.

`abstractmethod __fspath__()`

Renvoie la représentation du chemin du système de fichiers de l'objet.

La méthode ne devrait renvoyer que des objets *str* ou *bytes*, avec une préférence pour les *str*.

`os.getenv(key, default=None)`

Renvoie la valeur de la variable d'environnement `key` si elle existe, ou `default` si elle n'existe pas. `key`, `default`, et la valeur de retour sont des *str*.

Sur Unix, les clefs et les valeurs sont décodées avec `sys.getfilesystemencoding()` et le gestionnaire d'erreurs `surrogateescape`. Utilisez `os.getenvb()` si vous voulez utiliser un encodage différent.

Disponibilité : la plupart des dérivés Unix, Windows.

`os.getenvb(key, default=None)`

Renvoie la valeur de la variable d'environnement `key` si elle existe, ou `default` si elle n'existe pas. `key`, `default`, et la valeur de retour sont des *bytes*.

`getenvb()` n'est disponible que si `supports_byte_environ` vaut `True`.

Disponibilité : la plupart des dérivés Unix.

Nouveau dans la version 3.2.

os.**get_exec_path** (*env=None*)

Renvoie la liste des dossier qui seront parcouru pour trouver un exécutable, similaire à un shell lorsque il lance un processus. *env*, quand spécifié, doit être un dictionnaire de variable d'environnement afin d'y rechercher le PATH. Par défaut quand *env* est *None*, *environ* est utilisé.

Nouveau dans la version 3.2.

os.**getegid** ()

Renvoie l'identifiant du groupe effectif du processus actuel. Ça correspond au bit « *set id* » du fichier qui s'exécute dans le processus actuel.

Disponibilité : Unix.

os.**geteuid** ()

Renvoie l'identifiant de l'utilisateur effectif du processus actuel.

Disponibilité : Unix.

os.**getgid** ()

Renvoie l'identifiant de groupe réel du processus actuel.

Disponibilité : Unix.

os.**getgrouplist** (*user, group*)

Renvoie la liste d'identifiants de groupes auxquels *user* appartient. Si *group* n'est pas dans la liste, il y est inclus. Typiquement, *group* vaut le *group ID* de l'enregistrement *passwd* de *user*.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.**getgroups** ()

Renvoie une liste d'identifiants de groupes additionnels associés au processus actuel.

Disponibilité : Unix.

Note : Sur Mac OS X, le comportement de *getgroups()* diffère légèrement des autres plate-formes Unix. Si l'interpréteur Python était compilé avec une cible de déploiement de 10.5 ou moins, *getgroups()* renverrait la liste des identifiants de groupes effectifs associés au processus courant de l'utilisateur ; le nombre d'éléments de cette liste est limité par le système, typiquement 16, et peut être modifié par des appels à *setgroups()* si les privilèges ont été convenablement assignés. Si compilé avec une cible de déploiement supérieure à 10.5, la fonction *getgroups()* renvoie la liste des accès du groupe actuel pour l'utilisateur associé à l'identifiant utilisateur du processus ; la liste d'accès du groupe peut changer durant la vie du processus, elle n'est pas affectée par les appels à *setgroups()* et sa longueur n'est pas limitée à 16. La valeur de la cible de déploiement, *MACOSX_DEPLOYMENT_TARGET*, peut être obtenue par la fonction *sysconfig.get_config_var()*.

os.**getlogin** ()

Renvoie le nom de l'utilisateur connecté sur le terminal contrôlant le processus. Dans la plupart des cas, il est plus utile d'utiliser *getpass.getuser()* puisque cette fonction consulte les variables d'environnement *LOGNAME* et *USERNAME* pour savoir qui est l'utilisateur, et se replie finalement sur *pwd.getpwuid(os.getuid()) [0]* pour avoir le nom de connexion lié à l'identifiant de l'utilisateur courant.

Disponibilité : Unix, Windows.

os.**getpgid** (*pid*)

Renvoie l'identifiant de groupe de processus du processus de PID *pid*. Si *pid* vaut 0, l'identifiant de groupe de processus du processus actuel est renvoyé.

Disponibilité : Unix.

os.**getpgrp** ()

Renvoie l'identifiant du groupe de processus actuel.

Disponibilité : Unix.

`os.getpid()`

Renvoie l'identifiant du processus actuel.

`os.getppid()`

Renvoie l'identifiant du processus parent. Quand le processus parent est terminé, sur Unix, l'identifiant renvoyé est 1 pour le processus *init*, sur Windows, c'est toujours le même id, qui peut déjà avoir été réutilisé par un autre processus.

Disponibilité : Unix, Windows.

Modifié dans la version 3.2 : Prise en charge sur Windows.

`os.getpriority(which, who)`

Récupère la priorité d'ordonnancement des programmes. La valeur *which* est une des constantes suivantes : `PRIO_PROCESS`, `PRIO_PGRP`, ou `PRIO_USER`, et la valeur *who* est interprétée par rapport à *which* (un id de processus pour `PRIO_PROCESS`, un id de groupe de processus pour `PRIO_PGRP`, et un id d'utilisateur pour `PRIO_USER`). Une valeur nulle pour *who* dénote (respectivement) le processus appelant, le groupe de processus du processus appelant, ou l'identifiant d'utilisateur réel du processus appelant.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`os.PRIO_PROCESS`

`os.PRIO_PGRP`

`os.PRIO_USER`

Paramètres pour les fonctions `getpriority()` et `setpriority()`.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`os.getresuid()`

Renvoie un *tuple* (*ruid*, *euid*, *suid*) dénotant les identifiants de l'utilisateur réel, effectif et sauvé du processus actuel.

Disponibilité : Unix.

Nouveau dans la version 3.2.

`os.getresgid()`

Renvoie un *tuple* (*rgid*, *egid*, *sgid*) dénotant les identifiants des groupes de processus réel effectif, et sauvé du processus actuel.

Disponibilité : Unix.

Nouveau dans la version 3.2.

`os.getuid()`

Renvoie l'identifiant réel du processus actuel.

Disponibilité : Unix.

`os.initgroups(username, gid)`

Appelle la fonction système *initgroups* pour initialiser la liste des groupes d'accès des groupes dont *username* est membre, plus le groupe spécifié par *gid*.

Disponibilité : Unix.

Nouveau dans la version 3.2.

`os.putenv(key, value)`

Assigne la chaîne de caractères *value* à la variable d'environnement *key*. De tels changements à l'environnement affectent les sous-processus lancés par `os.system()`, `popen()` ou `fork()` et `execv()`.

Disponibilité : la plupart des dérivés Unix, Windows.

Note : Sur certaines plate-formes, incluant FreeBSD et Mac OS X, assigner `environ` peut causer des fuites de mémoire. Reférez-vous à la documentation système de `putenv`.

Quand `putenv()` est géré, les assignations d'éléments dans `os.environ` sont automatiquement traduites en appels correspondants à `putenv()`. Cependant, des appels à `putenv()` ne mettent pas `os.environ` à jour. Il est donc préférable d'assigner les éléments de `os.environ`.

os.setegid(*egid*)

Définit l'identifiant du groupe de processus effectif du processus actuel.

Disponibilité : Unix.

os.seteuid(*euid*)

Définit l'identifiant de l'utilisateur effectif du processus actuel.

Disponibilité : Unix.

os.setgid(*gid*)

Définit l'identifiant du groupe de processus actuel.

Disponibilité : Unix.

os.setgroups(*groups*)

Place *groups* dans la liste d'identifiants de groupes additionnels associée. *groups* doit être une séquence, et chaque élément doit être un nombre entier identifiant un groupe. Cette opération est typiquement disponible uniquement pour le super utilisateur.

Disponibilité : Unix.

Note : Sur Mac OS X, la longueur de *groups* ne peut pas dépasser le nombre maximum d'identifiants effectifs de groupes défini par le système, typiquement 16. Voir la documentation de `getgroups()` pour les cas où `getgroups` ne renvoie pas la même liste de groupes que celle définie par l'appel à `setgroups`.

os.setpgrp()

Produit l'appel système `setpgrp()` ou `setpgrp(0, 0)` selon la version implémentée (s'il y en a une). Voir le manuel Unix pour la sémantique de l'opération.

Disponibilité : Unix.

os.setpgid(*pid*, *pgrp*)

Produit l'appel système `setpgid()` pour placer l'identifiant du groupe de processus contenant le processus d'identifiant *pid* dans le groupe de processus d'identifiant *pgrp*. Voir le manuel Unix pour la sémantique.

Disponibilité : Unix.

os.setpriority(*which*, *who*, *priority*)

Définit la priorité d'ordonnancement des programmes. La valeur *which* est une des constantes suivantes : `PRIO_PROCESS`, `PRIO_PGRP`, ou `PRIO_USER`, et *who* est interprété en fonction de *which* (un PID pour `PRIO_PROCESS`, un identifiant de groupe de processus pour `PRIO_PGRP`, et un identifiant d'utilisateur pour `PRIO_USER`). Une valeur nulle pour *who* dénote (respectivement) le processus appelant, le groupe de processus du processus appelant, ou l'identifiant de l'utilisateur réel du processus appelant. *priority* est une valeur comprise entre -20 et 19. La priorité par défaut est 0 ; les priorités plus faibles amènent à un ordonnancement plus favorable.

Disponibilité : Unix

Nouveau dans la version 3.3.

os.setregid(*rgid*, *egid*)

Définit l'identifiant des groupes réel et effectif du processus actuel.

Disponibilité : Unix.

os.setresgid(*rgid*, *egid*, *sgid*)

Définit l'identifiant des groupes réel, effectif et sauvé du processus actuel.

Disponibilité : Unix.

Nouveau dans la version 3.2.

os.setresuid(*ruid*, *euid*, *suid*)

Définit l'identifiant des utilisateurs réel, effectif et sauvé du processus actuel.

Disponibilité : Unix.

Nouveau dans la version 3.2.

`os.setreuid(ruid, euid)`

Définit l'identifiant des utilisateurs réel et effectif du processus actuel.

Disponibilité : Unix.

`os.getsid(pid)`

Produit l'appel système `getsid()`. Voir le manuel Unix pour la sémantique.

Disponibilité : Unix.

`os.setsid()`

Produit l'appel système `setsid()`. Voir le manuel Unix pour la sémantique.

Disponibilité : Unix.

`os.setuid(uid)`

Définit l'identifiant de l'utilisateur du processus actuel.

Disponibilité : Unix.

`os.strerror(code)`

Renvoie le message d'erreur correspondant au code d'erreur *code*. Sur les plate-formes où `strerror()` renvoie `NULL` quand un numéro d'erreur inconnu est donné, une *ValueError* est levée.

`os.supports_bytes_environ`

`True` si le type natif de l'environnement du système d'exploitation est *bytes* (par exemple : `False` sur Windows).

Nouveau dans la version 3.2.

`os.umask(mask)`

Définit le *umask* actuel et renvoie la valeur précédente.

`os.uname()`

Renvoie des informations identifiant le système d'exploitation actuel. La valeur de retour est un objet à cinq attributs :

- `sysname` — nom du système d'exploitation
- `nodename` — nom de la machine sur le réseau (dépendant de l'implémentation)
- `release` — *release* du système d'exploitation
- `version` — version du système d'exploitation
- `machine` — identifiant du matériel

Pour la rétrocompatibilité, cet objet est également itérable, se comportant comme un quintuplet contenant `sysname`, `nodename`, `release`, `version`, et `machine` dans cet ordre.

Certains systèmes tronquent `nodename` à 8 caractères ou à la composante dominante. Un meilleur moyen de récupérer le nom de l'hôte est `socket.gethostname()` ou encore `socket.gethostbyaddr(socket.gethostname())`.

Disponibilité : dérivés récents de Unix.

Modifié dans la version 3.3 : Type de retour changé d'un *tuple* en un objet compatible avec le type *tuple*, avec des attributs nommés.

`os.unsetenv(key)`

Supprime la variable d'environnement appelée *key*. De tels changements à l'environnement affectent les sous-processus lancés avec `os.system()`, `popen()` ou `fork()` et `execv()`.

Quand `unsetenv()` est gérée, la suppression d'éléments dans `os.environ` est automatiquement interprétée en un appel correspondant à `unsetenv()`, mais les appels à `unsetenv()` ne mettent pas `os.environ` à jour. Donc il est préférable de supprimer les éléments de `os.environ`.

Disponibilité : la plupart des dérivés Unix, Windows.

16.1.3 Création de fichiers objets

Cette fonction crée de nouveaux *fichiers objets*. (Voir aussi `open()` pour ouvrir des descripteurs de fichiers).

`os.fdupen(fd, *args, **kwargs)`

Renvoie un fichier objet ouvert connecté au descripteur de fichier `fd`. C'est un alias de la primitive `open()` et accepte les mêmes arguments. La seule différence est que le premier argument de `fdopen()` doit toujours être un entier.

16.1.4 Opérations sur les descripteurs de fichiers

Ces fonctions opèrent sur des flux d'entrées/sorties référencés par des descripteurs de fichiers.

Les descripteurs de fichiers sont de petits entiers correspondant à un fichier qui a été ouvert par le processus courant. Par exemple, l'entrée standard est habituellement le descripteur de fichier 0, la sortie standard est 1, et le flux standard d'erreur est 2. Les autres fichiers ouverts par un processus vont se voir assigner 3, 4, 5, etc. Le nom « descripteur de fichier » est légèrement trompeur : sur les plate-formes Unix, les connecteurs (*socket* en anglais) et les tubes (*pipe* en anglais) sont également référencés par des descripteurs.

La méthode `fileno()` peut être utilisée pour obtenir le descripteur de fichier associé à un *file object* quand nécessaire. Notez qu'utiliser le descripteur directement outrepassse les méthodes du fichier objet, ignorant donc des aspects tels que la mise en tampon interne des données.

`os.close(fd)`

Ferme le descripteur de fichier `fd`.

Note : Cette fonction est destinée aux opérations d'entrées/sorties de bas niveau et doit être appliquée à un descripteur de fichier comme ceux donnés par `os.open()` ou `pipe()`. Pour fermer un « fichier objet » renvoyé par la primitive `open()`, `popen()` ou `fdopen()`, il faut utiliser sa méthode `close()`.

`os.closerange(fd_low, fd_high)`

Ferme tous les descripteurs de fichiers entre `fd_low` (inclus) jusqu'à `fd_high` (exclus), en ignorant les erreurs. Équivalent (mais beaucoup plus rapide) à :

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

`os.device_encoding(fd)`

Renvoie une chaîne de caractères décrivant l'encodage du périphérique associé à `fd` s'il est connecté à un terminal, sinon renvoie `None`.

`os.dup(fd)`

Renvoie une copie du descripteur de fichier `fd`. Le nouveau descripteur de fichier est *non-héritable*.

Sur Windows, quand un flux standard est dupliqué (0 : `stdin`, 1 : `stdout`, 2 : `stderr`), le nouveau descripteur de fichier est *héritable*.

Modifié dans la version 3.4 : Le nouveau descripteur de fichier est maintenant non-héritable.

`os.dup2(fd, fd2, inheritable=True)`

Copie le descripteur de fichier `fd` dans `fd2`, en fermant le premier si nécessaire. Le descripteur de fichier `fd2` est *héritable* par défaut, ou non-héritable si `inheritable` vaut `False`.

Modifié dans la version 3.4 : Ajout du paramètre optionnel `inheritable`.

-
- os.fchmod** (*fd*, *mode*)
 Change le mode du fichier donné par *fd* en la valeur numérique *mode*. Voir la documentation de `chmod()` pour les valeurs possibles de *mode*. Depuis Python 3.3, c'est équivalent à `os.chmod(fd, mode)`.
 Disponibilité : Unix.
- os.fchown** (*fd*, *uid*, *gid*)
 Change le propriétaire et l'identifiant de groupe du fichier donné par *fd* en les valeurs numériques *uid* et *gid*. Pour laisser l'un des identifiants inchangés, mettez-le à -1. Voir `chown()`. Depuis Python 3.3, c'est équivalent à `os.chown(fd, uid, gid)`.
 Disponibilité : Unix.
- os.fdatasync** (*fd*)
 Force l'écriture du fichier ayant le descripteur *fd* sur le disque. Ne force pas la mise à jour des méta-données.
 Disponibilité : Unix.
-
- Note :** Cette fonction n'est pas disponible sur MacOS.
-
- os.fpathconf** (*fd*, *name*)
 Renvoie les informations de la configuration du système pour un fichier ouvert. *name* spécifie la valeur de la configuration à récupérer, ça peut être une chaîne de caractères avec le nom d'une valeur système définie ; ces valeurs sont spécifiées dans certains standards (POSIX.1, Unix 95, Unix 98, et d'autres). Certaines plate-formes définissent des noms additionnels également. Les noms connus par le système d'exploitation sont donnés dans le dictionnaire `pathconf_names`. Pour les variables de configuration qui ne sont pas incluses dans ce *mapping*, passer un entier pour *name* est également accepté.
 Si *name* est une chaîne de caractères et n'est pas connu, une `ValueError` est levée. Si une valeur spécifique de *name* n'est pas gérée par le système hôte, même si elle est incluse dans `pathconf_names`, une `OSError` est levée avec `errno.EINVAL` pour code d'erreur.
 Depuis Python 3.3, c'est équivalent à `os.pathconf(fd, name)`.
 Disponibilité : Unix.
- os.fstat** (*fd*)
 Récupère le statut du descripteur de fichier *fd*. Renvoie un objet `stat_result`.
 Depuis Python 3.3, c'est équivalent à `os.stat(fd)`.
Voir aussi :
 La fonction `stat()`.
- os.fstatvfs** (*fd*)
 Renvoie des informations sur le système de fichier contenant le fichier associé au descripteur *fd*, comme `statvfs()`. Depuis Python 3.3, c'est équivalent à `os.statvfs(fd)`.
 Disponibilité : Unix.
- os.fsync** (*fd*)
 Force l'écriture du fichier ayant le descripteur *fd* sur le disque. Sur Unix, cet appel appelle la fonction native `fsync()`, sur Windows, la fonction `MS_commit()`.
 Si vous débutez avec un *file object* Python mis en tampon *f*, commencez par faire `f.flush()` et seulement ensuite `os.fsync(f.fileno())` pour être sûr que tous les tampons internes associés à *f* soient écrits sur le disque.
 Disponibilité : Unix, Windows.
- os.ftruncate** (*fd*, *length*)
 Tronque le fichier correspondant au descripteur *fd* pour qu'il soit maximum long de *length* bytes. Depuis Python 3.3, c'est équivalent à `os.truncate(fd, length)`.
 Disponibilité : Unix, Windows.
 Modifié dans la version 3.5 : Prise en charge de Windows

`os.get_blocking(fd)`

Récupère le mode bloquant du descripteur de fichier : `False` si l'indicateur `O_NONBLOCK` est mis, et `True` si l'indicateur est effacé.

Voir également `set_blocking()` et `socket.socket.setblocking()`.

Disponibilité : Unix.

Nouveau dans la version 3.5.

`os.isatty(fd)`

Renvoie `True` si le descripteur de fichier `fd` est ouvert et connecté à un périphérique TTY (ou compatible), sinon `False`.

`os.lockf(fd, cmd, len)`

Applique, teste, ou retire un verrou POSIX sur un descripteur de fichier ouvert. `fd` est un descripteur de fichier ouvert. `cmd` spécifie la commande à utiliser (une des valeurs suivantes : `F_LOCK`, `F_TLOCK`, `F_ULOCK`, ou `F_TEST`). `len` spécifie la section du fichier à verrouiller.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`os.F_LOCK`

`os.F_TLOCK`

`os.F_ULOCK`

`os.F_TEST`

Indicateurs spécifiant quelle action `lockf()` va prendre.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`os.lseek(fd, pos, how)`

Place la position actuelle du descripteur de fichier `fd` à la position `pos`, modifié par `how` : `SEEK_SET` ou 0 pour placer la position à partir du début du fichier, `SEEK_CUR` ou 1 pour la placer à partir de la position actuelle, et `SEEK_END` ou 2 pour la placer par rapport à la fin du fichier. Renvoie la nouvelle position du curseur en bytes, à partir du début.

`os.SEEK_SET`

`os.SEEK_CUR`

`os.SEEK_END`

Paramètres de la fonction `lseek()`. Leur valeur est respectivement 0, 1, et 2.

Nouveau dans la version 3.3 : Certains systèmes d'exploitation pourraient gérer des valeurs additionnelles telles que

`os.SEEK_HOLE` ou `os.SEEK_DATA`.

`os.open(path, flags, mode=0o777, *, dir_fd=None)`

Ouvre le fichier `path` et met certains indicateurs selon `flags` et éventuellement son mode selon `mode`. Lors de l'évaluation de `code`, ce `umask` actuel est d'abord masquée. Renvoie le descripteur de fichier du fichier nouvellement ouvert. Le nouveau descripteur de fichier est *non-héritable*.

Pour une description des indicateurs et des valeurs des modes, voir la documentation de la bibliothèque standard du C. Les constantes d'indicateurs (telles que `O_RDONLY` et `O_WRONLY`) sont définies dans le module `os`. En particulier, sur Windows, ajouter `O_BINARY` est nécessaire pour ouvrir des fichiers en binaire.

Cette fonction prend en charge des *chemins relatifs à des descripteurs de répertoires* avec le paramètre `dir_fd`.

Modifié dans la version 3.4 : Le nouveau descripteur de fichier est maintenant non-héritable.

Note : Cette fonction est destinée aux E/S de bas-niveau. Pour un usage normal, utilisez la primitive `open()` qui renvoie un *file object* avec les méthodes `read()` et `write()` (et plein d'autres). Pour envelopper un descripteur de fichier dans un fichier objet, utilisez `fdopen()`.

Nouveau dans la version 3.3 : L'argument `dir_fd`.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

Modifié dans la version 3.6 : Accepte un *path-like object*.

Les constantes suivantes sont optionnelles pour le paramètre *flag* à la fonction `open()`. Elles peuvent être combinées en utilisant l'opérateur bit-à-bit OR `|`. Certains ne sont pas disponibles sur toutes les plate-formes. Pour une description sur leur disponibilité et leur usage, consultez la page de manuel Unix `open(2)` ou la [MSDN](#) sur Windows.

```
os.O_RDONLY
os.O_WRONLY
os.O_RDWR
os.O_APPEND
os.O_CREAT
os.O_EXCL
os.O_TRUNC
```

Les constantes ci-dessus sont disponibles sur Unix et Windows.

```
os.O_DSYNC
os.O_RSYNC
os.O_SYNC
os.O_NDELAY
os.O_NONBLOCK
os.O_NOCTTY
os.O_CLOEXEC
```

Les constantes ci-dessus sont uniquement disponibles sur Unix.

Modifié dans la version 3.3 : Ajout de la constante `O_CLOEXEC`.

```
os.O_BINARY
os.O_NOINHERIT
os.O_SHORT_LIVED
os.O_TEMPORARY
os.O_RANDOM
os.O_SEQUENTIAL
os.O_TEXT
```

Les constantes ci-dessus sont uniquement disponibles sur Windows.

```
os.O_ASYNC
os.O_DIRECT
os.O_DIRECTORY
os.O_NOFOLLOW
os.O_NOATIME
os.O_PATH
os.O_TMPFILE
os.O_SHLOCK
os.O_EXLOCK
```

Les constantes ci-dessus sont des extensions et ne sont pas présentes si elles ne sont pas définies par la bibliothèque C.

Modifié dans la version 3.4 : Ajout de `O_PATH` sur les systèmes qui le gèrent. Ajout de `O_TMPFILE`, uniquement disponible sur Linux Kernel 3.11 ou plus récent.

```
os.openpty()
```

Ouvre une nouvelle paire pseudo-terminal. Renvoie une paire de descripteurs de fichiers (`master`, `slave`) pour le PTY et le TTY respectivement. Les nouveaux descripteurs de fichiers sont *non-héritables*. Pour une approche (légèrement) plus portable, utilisez le module `pty`.

Disponibilité : certains dérivés Unix.

Modifié dans la version 3.4 : Les nouveaux descripteurs de fichiers sont maintenant non-héritables.

os.**pipe**()

Crée un *pipe* (un tube). Renvoie une paire de descripteurs de fichiers (*r*, *w*) utilisables respectivement pour lire et pour écrire. Les nouveaux descripteurs de fichiers sont *non-héritables*.

Disponibilité : Unix, Windows.

Modifié dans la version 3.4 : Les nouveaux descripteurs de fichiers sont maintenant non-héritables.

os.**pipe2**(*flags*)

Crée un *pipe* avec *flags* mis atomiquement. *flags* peut être construit en appliquant l'opérateur | (OU) sur une ou plus de ces valeurs : *O_NONBLOCK*, *O_CLOEXEC*. Renvoie une paire de descripteurs de fichiers (*r*, *w*) utilisables respectivement pour lire et pour écrire.

Disponibilité : certains dérivés Unix.

Nouveau dans la version 3.3.

os.**posix_fallocate**(*fd*, *offset*, *len*)

Assure que suffisamment d'espace sur le disque est alloué pour le fichier spécifié par *fd* partant de *offset* et continuant sur *len* bytes.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.**posix_fadvise**(*fd*, *offset*, *len*, *advice*)

Annonce une intention d'accéder à des données selon un motif spécifique, et donc permettant au noyau de faire des optimisations. Le conseil *advice* s'applique à la région spécifiée par *fd*, démarrant à *offset* et continuant sur *len* bytes. *advice* est une des valeurs suivantes : *POSIX_FADV_NORMAL*, *POSIX_FADV_SEQUENTIAL*, *POSIX_FADV_RANDOM*, *POSIX_FADV_NOREUSE*, *POSIX_FADV_WILLNEED*, ou *POSIX_FADV_DONTNEED*.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.**POSIX_FADV_NORMAL**

os.**POSIX_FADV_SEQUENTIAL**

os.**POSIX_FADV_RANDOM**

os.**POSIX_FADV_NOREUSE**

os.**POSIX_FADV_WILLNEED**

os.**POSIX_FADV_DONTNEED**

Indicateurs qui peuvent être utilisés dans *advice* dans la fonction *posix_fadvise()* et qui spécifient le motif d'accès qui est censé être utilisé.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.**pread**(*fd*, *bufferize*, *offset*)

Lit depuis un descripteur de fichier *fd* à la position de l'*offset*. Cela va lire jusqu'à *bufferize* bytes. L'offset du fichier ne sera pas changé.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.**pwrite**(*fd*, *str*, *offset*)

Écrit *bytestring* dans un descripteur de fichier *fd* à une position *offset* en laissant le décalage du fichier inchangé.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.**read**(*fd*, *n*)

Lit au plus *n* bytes depuis le descripteur *fd*. Renvoie une chaîne de bytes contenant les bytes lus. Si la fin du fichier référencé par *fd* est atteinte, un objet vide de bytes est renvoyé.

Note : Cette fonction est destinée aux E/S bas niveau et doit être appliquée à un descripteur de fichier comme renvoyé par `os.open()` ou `pipe()`. Pour lire dans un « fichier objet » renvoyé par la primitive `open()`, `popen()` ou `fdopen()`, ou par `stdin`, utilisez sa méthode `read()` ou `readline()`.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

`os.sendfile(out, in, offset, count)`

`os.sendfile(out, in, offset, count[, headers][, trailers], flags=0)`

Copie *count* bytes depuis un descripteur de fichier *in* dans un descripteur de fichier *out* en démarrant à *offset*. Renvoie le nombre de bytes envoyés. Quand EOF est atteint, renvoie 0.

La première notation de fonction est prise en charge par toutes les plate-formes qui définissent `sendfile()`.

Sur Linux, si *offset* est donné par `None`, les bytes sont lus depuis la position actuelle de *in* et la position de *in* est mise à jour.

Le second cas peut être utilisé sur Mac OS X et FreeBSD où *headers* et *trailers* sont des séquences arbitraires de tampons qui sont écrites avant et après que les données de *in* ne soient écrites. Renvoie la même chose que le premier cas.

Sur Mac OS X et FreeBSD, une valeur de 0 pour *count* spécifié d'envoyer jusqu'à ce que la fin de *in* ne soit atteinte.

Toutes les plate-formes gèrent les connecteurs comme des descripteurs de fichier *out*, et certaines plate-formes autorisent d'autres types (par exemple, un fichier normal ou un tube) également.

Les applications multiplate-formes ne devraient pas utiliser les arguments *headers*, *trailers*, et *flags*.

Disponibilité : Unix.

Note : Pour une interface de plus haut niveau de `sendfile()`, voir `socket.socket.setfile()`.

Nouveau dans la version 3.3.

`os.set_blocking(fd, blocking)`

Définit le mode bloquant d'un descripteur de fichier spécifié. Assigne l'indicateur `O_NONBLOCK` si *blocking* vaut `False`, efface l'indicateur sinon.

Voir aussi `get_blocking()` et `socket; socket.setblocking()`.

Disponibilité : Unix.

Nouveau dans la version 3.5.

`os.SF_NODISKIO`

`os.SF_MNOWAIT`

`os.SF_SYNC`

Paramètres de la fonction `sendfile()`, si l'implémentation les gère.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`os.readv(fd, buffers)`

Lit depuis un descripteur de fichier *fd* dans un nombre d'objets *bytes-compatibles* muables : *buffers*. `readv()` va transférer les données dans chaque tampon jusqu'à ce qu'il soit plein et puis passer au tampon suivant dans la séquence pour y placer le reste de la donnée. `readv()` renvoie le nombre total de bytes lus (qui peut être moins que la capacité totale de tous les objets).

Disponibilité : Unix.

Nouveau dans la version 3.3.

`os.tcgetpgrp(fd)`

Renvoie le groupe de processus associé au terminal donné par *fd* (un descripteur de fichier ouvert comme renvoyé par `os.open()`).

Disponibilité : Unix.

`os.tcsetpgrp(fd, pg)`

Place *pg* dans le groupe de processus associé au terminal donné par *fd* (un descripteur de fichier ouvert comme renvoyé par `os.open()`).

Disponibilité : Unix.

`os.ttyname(fd)`

Renvoie une chaîne de caractères spécifiant le périphérique terminal associé au descripteur de fichier *fd*. Si *fd* n'est pas associé à un périphérique terminal, une exception est levée.

Disponibilité : Unix.

`os.write(fd, str)`

Écrit la chaîne de bytes de *str* dans le descripteur de fichier *fd*. Renvoie le nombre de bytes réellement écrits.

Note : Cette fonction est destinée aux entrées-sorties bas niveau et doit être appliquée à un descripteur de fichier comme renvoyé par `os.open()` ou `pipe()`. Pour écrire dans un « fichier objet » renvoyé par la primitive `open()`, `popen()`, ou par `fdopen()`, ou par `sys.stdout` ou `sys.stderr`, utilisez sa méthode `write()`.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

`os.writev(fd, buffers)`

Écrit le contenu de *buffers* vers le descripteur de fichier *fd*. *buffers* doit être une séquence d'*objets bytes-compatibles*. Les tampons sont traités dans l'ordre du tableau. Le contenu entier du premier tampon est écrit avant le traitement du second, etc. Le système d'exploitation peut avoir une limite (valeur `sysconf()` `SC_IOV_MAX`) sur le nombre de tampons qui peuvent être utilisés.

`writev()` écrit le contenu de chaque objet vers le descripteur de fichier et renvoie le nombre total d'octets écrits.

Disponibilité : Unix.

Nouveau dans la version 3.3.

Demander la taille d'un terminal

Nouveau dans la version 3.3.

`os.get_terminal_size(fd=STDOUT_FILENO)`

Renvoie la taille du terminal comme un couple (`columns`, `lines`) de type `terminal_size`.

L'argument optionnel *fd* (par défaut : `STDOUT_FILENO`, ou la sortie standard) spécifie le descripteur de fichier auquel la requête doit être envoyée.

Si le descripteur de fichier n'est pas connecté à un terminal, une `OSError` est levée.

`shutil.get_terminal_size()` est la fonction haut-niveau qui devrait normalement être utilisée, `os.get_terminal_size` en est l'implémentation bas-niveau.

Disponibilité : Unix, Windows.

class `os.terminal_size`

Une sous-classe de `tuple`, contenant (`columns`, `lines`), la taille du terminal.

columns

Longueur du terminal en caractères.

lines

Hauteur du terminal en caractères.

Héritage de descripteurs de fichiers

Nouveau dans la version 3.4.

Un descripteur de fichier a un indicateur indiquant s'il peut être hérité par les processus-fils. Depuis Python 3.4, les descripteurs de fichiers créés par Python ne sont pas héritables par défaut.

Sur UNIX, les descripteurs de fichiers non-héritables sont fermés dans les processus-fils à l'exécution, les autres descripteurs sont hérités.

Sur Windows, les fichiers et identificateurs non-héritables sont fermés dans les processus-fils, à part les flux standards (descripteurs 0, 1, et 2 : *stdin*, *stdout* et *stderr*) qui sont toujours hérités. En utilisant les fonctions *spawn**, tous les identificateurs héritables et les descripteurs de fichiers héritables sont hérités. En utilisant le module *subprocess*, tous les descripteurs de fichiers (à part les flux standards) sont fermés, et les identificateurs héritables sont hérités seulement si le paramètre *close_fds* vaut *False*.

- `os.get_inheritable(fd)`
Récupère le marqueur « héritable » (booléen) du descripteur de fichier spécifié.
- `os.set_inheritable(fd, inheritable)`
Définit le marqueur « héritable » du descripteur de fichier spécifié.
- `os.get_handle_inheritable(handle)`
Récupère le marqueur « héritable » (booléen) de l'identificateur spécifié.
Disponibilité : Windows.
- `os.set_handle_inheritable(handle, inheritable)`
Définit le marqueur « héritable » de l'identificateur spécifié.
Disponibilité : Windows.

16.1.5 Fichiers et répertoires

Sur certaines plate-formes Unix, beaucoup de ces fonctions gèrent une ou plusieurs des fonctionnalités suivantes :

- **spécifier un descripteur de fichier** : pour certaines fonctions, l'argument *path* peut être non seulement une chaîne de caractères donnant le chemin vers le fichier, mais également un descripteur de fichier. La fonction opérera alors sur le fichier référencé par le descripteur. (Pour les systèmes POSIX, Python appellera la version `f...at` de la fonction.)
Vous pouvez vérifier si *path* peut être donné par un descripteur de fichier sur votre plate-forme en utilisant `os.supports_fd`. Si c'est indisponible, l'utiliser lèvera une *NotImplementedError*.
Si la fonction gère également les arguments *dir_fd* ou *follow_symlinks*, spécifier l'un de ces arguments est une erreur quand *path* est donné en tant que descripteur de fichier.
 - **Chemins relatifs vers des descripteurs de répertoires** : si *dir_fd* n'est pas *None*, il devrait être un descripteur de fichier référençant un répertoire, et le chemin sur lequel opérer devrait être relatif. Le chemin est donc relatif à ce répertoire. Si le chemin est absolu, *dir_fd* est ignoré. (Pour les systèmes POSIX, Python appellera la version `...at` ou `f...at` de la fonction.)
Vous pouvez vérifier si *dir_fd* est géré sur votre plate-forme en utilisant `os.supports_dir_fd`. Si c'est indisponible, l'utiliser lèvera une *NotImplementedError*.
 - **Non-suivi des symlinks (liens symboliques)** : si *follow_symlinks* vaut *False* et le dernier élément du chemin sur lequel opérer est un lien symbolique, la fonction opérera sur le lien symbolique et pas sur le fichier pointé par le lien. (Pour les systèmes POSIX, Python appellera la version `l...at` de la fonction.)
Vous pouvez voir si *follow_symlinks* est géré sur votre plate-forme en utilisant `os.supports_follow_symlinks`. Si c'est indisponible, l'utiliser lèvera une *NotImplementedError*.
- `os.access(path, mode, *, dir_fd=None, effective_ids=False, follow_symlinks=True)`
Utilise l'*uid/gid* réel pour tester l'accès à *path*. Notez que la plupart des opérations utiliseront l'*uid/gid* effectif,

dès lors cette méthode peut être utilisée dans un environnement *suid/sgid* pour tester si l'utilisateur invoquant a les droits d'accès pour accéder à *path*. *mode* devrait être *F_OK* pour tester l'existence de *path*, ou il peut être le OR (OU inclusif) d'une ou plusieurs des constantes suivantes : *R_OK*, *W_OK*, et *X_OK* pour tester les permissions. Renvoie *True* si l'accès est permis, et *False* s'il ne l'est pas. Voir la page de manuel Unix *access(2)* pour plus d'informations.

Cette fonction peut gérer la spécification de *chemins relatifs vers des descripteurs de fichiers* et *le suivi des liens symboliques*.

Si *effective_id* vaut *True*, *access()* effectuera ses vérifications d'accès en utilisant l'*uid/gid* effectif à la place de l'*uid/gid* réel. *effective_ids* peut ne pas être géré sur votre plate-forme, vous pouvez vérifier s'il est disponible en utilisant *os.supports_effective_ids*. S'il est indisponible, l'utiliser lèvera une *NotImplementedError*.

Note : Utiliser *access()* pour vérifier si un utilisateur est autorisé (par exemple) à ouvrir un fichier avant d'effectivement le faire en utilisant *open()* crée une faille de sécurité : l'utilisateur peut exploiter le court intervalle de temps entre la vérification et l'ouverture du fichier pour le manipuler. Il est préférable d'utiliser les techniques *EAFP*. Par exemple :

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

est mieux écrit comme suit :

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

Note : Les opérations d'entrées/sorties peuvent échouer même quand *access()* indique qu'elles devraient réussir, particulièrement pour les opérations sur les systèmes de fichiers réseaux qui peuvent avoir une sémantique de permissions au-delà du modèle de bits de permission usuel POSIX.

Modifié dans la version 3.3 : Paramètres *dir_fd*, *effective_ids*, et *follow_symlinks* ajoutés.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.F_OK`
`os.R_OK`
`os.W_OK`
`os.X_OK`

Valeurs à passer au paramètre *mode* de *access()* pour tester respectivement l'existence, les droits de lecture, d'écriture et d'exécution.

`os.chdir(path)`

Change le répertoire de travail actuel par *path*.

Cette fonction prend en charge *la spécification d'un descripteur de fichier*. Le descripteur doit référencer un répertoire ouvert, pas un fichier ouvert.

Nouveau dans la version 3.3 : Prise en charge de la spécification de *path* par un descripteur de fichier sur certaines plate-formes.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.chflags(path, flags, *, follow_symlinks=True)`

Définit les marqueurs de *path* par la valeur numérique *flags*. *flags* peut prendre une combinaison (OU bit-à-bit) des valeurs suivantes (comme défini dans le module *stat*) :

- `stat.UF_NODUMP`
- `stat.UF_IMMUTABLE`
- `stat.UF_APPEND`
- `stat.UF_OPAQUE`
- `stat.UF_NOUNLINK`
- `stat.UF_COMPRESSED`
- `stat.UF_HIDDEN`
- `stat.SF_ARCHIVED`
- `stat.SF_IMMUTABLE`
- `stat.SF_APPEND`
- `stat.SF_NOUNLINK`
- `stat.SF_SNAPSHOT`

Cette fonction prend en charge *le suivi des liens symboliques*.

Disponibilité : Unix.

Nouveau dans la version 3.3 : L'argument *follow_symlinks*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.chmod(path, mode, *, dir_fd=None, follow_symlinks=True)`

Change le mode de *path* par la valeur numérique *mode*. *mode* peut prendre une des valeurs suivantes (comme défini dans le module *stat*) ou une combinaison (OU bit-à-bit) de ces valeurs :

- `stat.S_ISUID`
- `stat.S_ISGID`
- `stat.S_ENFMT`
- `stat.S_ISVTX`
- `stat.S_IREAD`
- `stat.S_IWRITE`
- `stat.S_IEXEC`
- `stat.S_IRWXU`
- `stat.S_IRUSR`
- `stat.S_IWUSR`
- `stat.S_IXUSR`
- `stat.S_IRWXG`
- `stat.S_IRGRP`
- `stat.S_IWGRP`
- `stat.S_IXGRP`
- `stat.S_IRWXO`
- `stat.S_IROTH`
- `stat.S_IWOTH`
- `stat.S_IXOTH`

Cette fonction prend en charge *la spécification d'un descripteur de fichier, les chemins relatifs à des descripteurs de répertoires, et le non-suivi des liens symboliques*.

Note : Bien que Windows gère `chmod()`, vous ne pouvez y définir que le marqueur de lecture-seule du fichier (via les constantes `stat.S_IWRITE` et `stat.S_IREAD` ou une constante entière correspondante). Tous les autres bits sont ignorés.

Nouveau dans la version 3.3 : Prise en charge de la spécification de *path* par un répertoire ouvert et des arguments *dir_fd* et *follow_symlinks* ajoutés.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.chown(path, uid, gid, *, dir_fd=None, follow_symlinks=True)`

Change l'identifiant du propriétaire et du groupe de *path* par les valeurs numériques *uid* et *gid*. Pour laisser l'un de ces identifiants inchangé, le définir à `-1`.

Cette fonction prend en charge *la spécification d'un descripteur de fichier, les chemins relatifs à des descripteurs de répertoires, et le non-suivi des liens symboliques*.

Voir `shutil.chown()` pour une fonction de plus haut-niveau qui accepte des noms en plus des identifiants numériques.

Disponibilité : Unix.

Nouveau dans la version 3.3 : Prise en charge de la spécification de *path* par un descripteur de fichier ouvert et des arguments *dir_fd* et *follow_symlinks* ajoutés.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.chroot(*path*)

Change le répertoire racine du processus actuel par *path*.

Disponibilité : Unix.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.fchdir(*fd*)

Change le répertoire de travail actuel par le répertoire représenté par le descripteur de fichier *fd*. Le descripteur doit référencer un répertoire ouvert, pas un fichier ouvert. Depuis Python 3.3, c'est équivalent à `os.chdir(fd)`.

Disponibilité : Unix.

os.getcwd()

Renvoie une chaîne de caractères représentant le répertoire de travail actuel.

os.getcwdb()

Renvoie une chaîne de *bytes* représentant le répertoire de travail actuel.

os.lchflags(*path*, *flags*)

Définit les marqueurs de *path* par la valeur numérique *flags*, comme `chflags()`, mais ne suit pas les liens symboliques. Depuis Python 3.3, c'est équivalent à `os.chflags(path, flags, follow_symlinks=False)`.

Disponibilité : Unix.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.lchmod(*path*, *mode*)

Change le mode de *path* par la valeur numérique *mode*. Si *path* est un lien symbolique, ça affecte le lien symbolique à la place de la cible. Voir la documentation pour les valeurs possibles de *mode* pour `chmod()`. Depuis Python 3.3, c'est équivalent à `os.chmod(path, mode, follow_symlinks=False)`.

Disponibilité : Unix.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.lchown(*path*, *uid*, *gid*)

Change les identifiants du propriétaire et du groupe de *path* par *uid* et *gid*. Cette fonction ne suivra pas les liens symboliques. Depuis Python 3.3, c'est équivalent à `os.chown(path, uid, gid, follow_symlinks=False)`.

Disponibilité : Unix.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.link(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*, *follow_symlinks=True*)

Crée un lien matériel appelé *dst* pointant sur *src*.

Cette fonction prend en charge la spécification *src_dir_fd* et/ou *dst_dir_fd* pour préciser *des chemins relatifs à des descripteurs de répertoires, et le non-suivi des liens symboliques*.

Disponibilité : Unix, Windows.

Modifié dans la version 3.2 : Prise en charge de Windows.

Nouveau dans la version 3.3 : Arguments *src_dir_fd*, *dst_dir_fd*, et *follow_symlinks* ajoutés.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *src* et *dst*.

os.listdir (*path*='.')

Renvoie une liste contenant le nom des entrées dans le répertoire donné par *path*. La liste est dans un ordre arbitraire et n'inclut pas les entrées spéciales `'.'` et `'..'` même si elles sont présentes dans le répertoire.

path peut être un *path-like object*. Si *path* est de type `bytes` (directement ou indirectement à travers une interface *PathLike*), les noms de fichiers renvoyés seront aussi de type `bytes`; dans toutes les autres circonstances, ils seront de type `str`.

Cette fonction peut également gérer *la spécification de descripteurs de fichiers*. Le descripteur doit référencer un répertoire.

Note : Pour encoder des noms de fichiers de type `str` en `bytes`, utilisez la fonction `encode()`.

Voir aussi :

La fonction `scandir()` renvoie les entrées du répertoire ainsi que leurs attributs, offrant une meilleure performance pour beaucoup de cas utilisés fréquemment.

Modifié dans la version 3.2 : Le paramètre *path* est devenu optionnel.

Nouveau dans la version 3.3 : Prise en charge de la spécification d'un descripteur de répertoire pour *path* ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.lstat (*path*, *, *dir_fd*=None)

Effectue l'équivalent d'un appel système `lstat()` sur le chemin donné. Similaire à `stat()` mais ne suit pas les liens symboliques. Renvoie un objet de type `stat_result`.

Sur les plate-formes qui ne gèrent pas les liens symboliques, c'est un alias pour `stat()`.

Depuis Python 3.3, c'est équivalent à `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`.

Cette fonction peut également gérer *des chemins relatifs à des descripteurs de répertoires*.

Voir aussi :

La fonction `stat()`.

Modifié dans la version 3.2 : Prise en charge des liens symboliques sur Windows 6.0 (Vista).

Modifié dans la version 3.3 : Paramètre *dir_fd* ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *src* et *dst*.

os.mkdir (*path*, *mode*=0o777, *, *dir_fd*=None)

Crée un répertoire appelé *path* avec pour *mode*, la valeur numérique *mode*.

Si le répertoire existe déjà, `FileExistsError` est levée.

Sous certains systèmes, *mode* est ignoré. Quand il est utilisé, il lui est premièrement appliqué le masque courant *umask*. Si des bits autres que les 9 derniers sont activés (i.e. les 3 derniers chiffres de la représentation octale de *mode*), leur signification sera dépendante de la plate-forme. Sous certaines plate-formes, ils seront ignorés et vous devrez appeler explicitement `chmod()` pour les modifier.

Cette fonction peut également gérer *des chemins relatifs à des descripteurs de répertoires*.

Il est également possible de créer des répertoires temporaires, voir la fonction `tempfile.mkdtemp()` du module `tempfile`.

Nouveau dans la version 3.3 : L'argument *dir_fd*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.makedirs (*name*, *mode*=0o777, *exist_ok*=False)

Fonction de création récursive de répertoires. Comme `mkdir()` mais crée tous les répertoires de niveau intermédiaire nécessaires pour contenir le répertoire « feuille ».

Le paramètre *mode* est passé à `mkdir()`; voir *la description de mkdir()* pour comprendre comment cela est interprété.

Si *exist_ok* vaut `False` (valeur par défaut), une `OSError` est levée si le répertoire cible existe déjà.

Note : Un appel à `makedirs()` est confus si les éléments du chemin à créer contiennent `pardir` (par exemple, « .. » sur les systèmes UNIX).

Cette fonction gère les chemins UNC correctement.

Nouveau dans la version 3.2 : Le paramètre `exist_ok`.

Modifié dans la version 3.4.1 : Avant Python 3.4.1, si `exist_ok` valait `True` et le répertoire à créer existait, `makedirs()` aurait levé une erreur si `mode` n'était pas équivalent au mode du répertoire existant. Puisque ce comportement n'était pas possible) implémenter de manière sécurisée, il a été retiré pour Python 3.4.1. Voir [bpo-21082](#).

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.**mkfifo** (*path*, *mode=0o666*, *, *dir_fd=None*)

Crée un FIFO (*First In, First Out*, ou un tube (*pipe* en anglais) nommé) appelé *path* avec le mode numérique *mode*. La valeur actuelle de *umask* est d'abord masquée du mode.

Cette fonction peut également gérer *des chemins relatifs à des descripteurs de répertoires*.

Les FIFOs sont des tubes qui peuvent être accédés comme des fichiers normaux. Les FIFOs existent jusqu'à ce qu'ils soient retirés (par exemple, à l'aide de `os.unlink()`). Généralement, les FIFOs sont utilisés comme communication entre des processus de type « client » et « serveur » : le serveur ouvre le FIFO pour le lire, et le client l'ouvre pour écrire dedans. Notez que `mkfifo()` n'ouvre pas le FIFO — il crée juste un point de rendez-vous.

Disponibilité : Unix.

Nouveau dans la version 3.3 : L'argument *dir_fd*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.**mknod** (*path*, *mode=0o600*, *device=0*, *, *dir_fd=None*)

Crée un nœud du système de fichiers (fichier, périphérique, fichier spécial, ou tuyau nommé) appelée *path*. *mode* spécifie à la fois les permissions à utiliser et le type de nœud à créer, en étant combiné (OR bit-à-bit) avec l'une des valeurs suivantes : `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, et `stat.S_IFIFO` (ces constantes sont disponibles dans le module `stat`). Pour `stat.S_IFCHR` et `stat.S_IFBLK`, *device* définit le fichier spécial de périphérique tout juste créé (probablement en utilisant `os.makedev()`), sinon, cet argument est ignoré.

Cette fonction peut également gérer *des chemins relatifs à des descripteurs de répertoires*.

Disponibilité : Unix.

Nouveau dans la version 3.3 : L'argument *dir_fd*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.**major** (*device*)

Extrait le nombre majeur de périphérique d'un nombre de périphérique brut (habituellement le champ `st_dev` ou `st_rdev` de `stat`).

os.**minor** (*device*)

Extrait le nombre mineur de périphérique d'un nombre de périphérique brut (habituellement le champ `st_dev` ou `st_rdev` de `stat`).

os.**makedev** (*major*, *minor*)

Compose un nombre de périphérique brut à partir des nombres de périphérique mineur et majeur.

os.**pathconf** (*path*, *name*)

Renvoie des informations sur la configurations relatives à un fichier déterminé. *name* spécifie la valeur de configuration à récupérer ; ce peut être une chaîne de caractères qui est le nom d'une valeur système particulière. Ces noms sont spécifiés dans certains standards (POSIX.1, Unix 95, Unix 98, etc). Certaines plate-formes définissent des noms supplémentaires également. Les noms connus du système d'exploitation hôte sont donnés dans le dictionnaire `pathconf_names`. Pour les variables de configuration non incluses dans ce *mapping*, passer un entier pour *name* est également accepté.

Si *name* est une chaîne de caractères et n'est pas connu, une `ValueError` est levée. Si une valeur spécifique de *name* n'est pas gérée par le système hôte, même si elle est incluse dans `pathconf_names`, une `OSError` est levée avec `errno.EINVAL` pour code d'erreur.

Cette fonction prend en charge *la spécification d'un descripteur de fichier*.

Disponibilité : Unix.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.pathconf_names

Dictionnaire liant les noms acceptés par les fonctions `pathconf()` et `fpathconf()` aux valeurs entières définies pour ces noms par le système d'exploitation hôte. Cette variable peut être utilisée pour déterminer l'ensemble des noms connus du système d'exploitation.

Disponibilité : Unix.

os.readlink(path, *, dir_fd=None)

Renvoie une chaîne de caractères représentant le chemin vers lequel le lien symbolique pointe. Le résultat peut être soit un chemin relatif, soit un chemin absolu. S'il est relatif, il peut être converti en chemin absolu en utilisant `os.path.join(os.path.dirname(path), result)`.

Si *path* est une chaîne de caractères (directement ou indirectement à travers une interface *PathLike*), le résultat sera aussi une chaîne de caractères, et l'appel pourra lever une `UnicodeDecodeError`. Si *path* est une chaîne d'octets (directement ou indirectement), le résultat sera une chaîne d'octets.

Cette fonction peut également gérer *des chemins relatifs à des descripteurs de répertoires*.

Disponibilité Unix, Windows

Modifié dans la version 3.2 : Prise en charge des liens symboliques sur Windows 6.0 (Vista).

Nouveau dans la version 3.3 : L'argument *dir_fd*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.remove(path, *, dir_fd=None)

Supprime (retire) le fichier représenté par *path*. Si *path* est un répertoire, une `OSError` est levée. Utilisez la fonction `rmdir()` pour supprimer un répertoire.

Cette fonction prend en charge *des chemins relatifs à des descripteurs de répertoires*.

Sur Windows, tenter de retirer un fichier en cours d'utilisation cause la levée d'une exception, sur Unix, l'entrée du répertoire est supprimé mais l'espace de stockage alloué au fichier ne sera pas disponible avant que le fichier original ne soit plus utilisé.

La fonction est sémantiquement identique à `unlink()`.

Nouveau dans la version 3.3 : L'argument *dir_fd*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.removedirs(name)

Supprime des répertoires récursivement. Fonctionne comme `rmdir()` si ce n'est que si le répertoire feuille est retiré avec succès, `removedirs()` essaye de supprimer successivement chaque répertoire parent mentionné dans *path* jusqu'à ce qu'un erreur ne soit levée (ce qui est ignoré car la signification générale en est qu'un répertoire parent n'est pas vide). Par exemple, `os.removedirs('foo/bar/baz')` supprimera d'abord le répertoire `'foo/bar/baz'`, et ensuite supprimera `'foo/bar'` et puis `'foo'` s'ils sont vides. Lève une `OSError` si le répertoire feuille n'a pas pu être supprimé avec succès.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None)

Renomme le fichier ou le répertoire *src* en *dst*. Si *dst* est un répertoire, une `OSError` est levée. Sur Unix, si *dst* existe, il sera remplacé silencieusement si l'utilisateur en a la permission. L'opération peut échouer sur certaines distributions Unix si *src* et *dst* sont sur des systèmes de fichiers séparés. Si le renommage est effectué avec succès, il est une opération atomique (nécessité POSIX). Sur Window, si *dst* existe déjà, une `OSError` est levée même s'il est un fichier.

Cette fonction prend en charge les spécifications *src_dir_fd* et/ou *dst_dir_fd* pour fournir *des chemins relatifs à des descripteurs de fichiers*.

Si vous désirez un écrasement multiplate-forme de la destination, utilisez la fonction `replace()`.

Nouveau dans la version 3.3 : Les arguments *src_dir_fd* et *dst_dir_fd*.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *src* et *dst*.

os.rename (*old, new*)

Fonction récursive de renommage de fichiers ou répertoires. Fonctionne comme `rename()`, si ce n'est que la création d'un répertoire intermédiaire nécessaire pour rendre le nouveau chemin correct est essayé en premier. Après le renommage, les répertoires correspondant aux segments de chemin les plus à droite de l'ancien nom seront élagués en utilisant `removedirs()`.

Note : Cette fonction peut échouer avec la nouvelle structure de dictionnaire définie si vous n'avez pas les permissions nécessaires pour supprimer le répertoire ou fichier feuille.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *old* et *new*.

os.replace (*src, dst, *, src_dir_fd=None, dst_dir_fd=None*)

Renomme le fichier ou le répertoire *src* en *dst*. Si *dst* est un répertoire, une `OSError` est levée. Si *dst* existe et est un fichier, il sera remplacé silencieusement si l'utilisateur en a les permissions. L'opération peut échouer si *src* et *dst* sont sur un système de fichiers différent. Si le renommage est effectué avec succès, il est une opération atomique (nécessité POSIX).

Cette fonction prend en charge les spécifications *src_dir_fd* et/ou *dst_dir_fd* pour fournir *des chemins relatifs à des descripteurs de fichiers*.

Nouveau dans la version 3.3.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *src* et *dst*.

os.rmdir (*path, *, dir_fd=None*)

Supprime (retire) le répertoire *path*. Ne fonctionne que lorsque le répertoire est vide, sinon une `OSError` est levée. Afin de supprimer toute la hiérarchie de dossier, la fonction `shutil.rmtree()` peut être utilisée.

Cette fonction prend en charge *des chemins relatifs à des descripteurs de répertoires*.

Nouveau dans la version 3.3 : Le paramètre *dir_fd*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.scandir (*path='.'*)

Renvoie un itérateur d'objets `os.DirEntry` correspondant aux entrées dans le répertoire indiqué par *path*. Les entrées sont produites dans un ordre arbitraire, et les entrées spéciales `'.'` et `'..'` ne sont pas incluses.

Utiliser `scandir()` plutôt que `listdir()` peut significativement améliorer les performances des codes qui nécessitent aussi l'accès aux types des fichiers ou à leurs attributs, puisque les objets `os.DirEntry` exposent ces informations si le système d'exploitation les fournit en scannant le répertoire. Toutes les méthodes de `os.DirEntry` peuvent réaliser un appel système, mais `is_dir()` et `is_file()` n'en requièrent normalement un que pour les liens symboliques ; `os.DirEntry.stat()` nécessite toujours un appel système sous Unix, mais seulement pour les liens symboliques sous Windows.

path peut être un *path-like object*. Si *path* est de type `bytes` (directement ou indirectement à travers une interface `PathLike`), le type des attributs *name* et *path* de chaque `os.DirEntry` sera `bytes` ; dans toutes les autres circonstances, ils seront de type `str`.

L'itérateur `scandir()` gère le protocole *context manager* et possède la méthode suivante :

`scandir.close()`

Ferme l'itérateur et libère les ressources acquises.

Elle est appelée automatiquement quand l'itérateur est entièrement consommé ou collecté par le ramasse-miettes, ou quand une erreur survient durant l'itération. Il est cependant conseillé de l'appeler explicitement ou d'utiliser l'instruction `with`.

Nouveau dans la version 3.6.

L'exemple suivant montre une utilisation simple de `scandir()` pour afficher tous les fichiers (à part les répertoires) dans le chemin donné par *path* et ne débutant pas par `'.'`. L'appel `entry.is_file()` ne va généralement pas faire d'appel système supplémentaire :

```
with os.scandir(path) as it:
    for entry in it:
```

(suite sur la page suivante)

(suite de la page précédente)

```
if not entry.name.startswith('.') and entry.is_file():
    print(entry.name)
```

Note : Sur les systèmes inspirés de Unix, `scandir()` utilise les fonctions système `opendir()` et `readdir()`. Sur Windows, la fonction utilise les fonctions Win32 `FindFirstFileW` et `FindNextFileW`.

Nouveau dans la version 3.5.

Nouveau dans la version 3.6 : Prise en charge du protocole *context manager* et de la méthode `close()`. Si un itérateur sur `scandir()` n'est ni entièrement consommé ni explicitement fermé, un `ResourceWarning` sera émis dans son destructeur.

La fonction accepte un *path-like object*.

class `os.DirEntry`

Objet donné par `scandir()` pour exposer le chemin du fichier et d'autres attributs de fichier d'une entrée du répertoire.

`scandir()` fournira autant d'informations que possible sans faire d'appels système additionnels. Quand un appel système `stat()` ou `lstat()` est réalisé, l'objet `os.DirEntry` mettra le résultat en cache.

Les instances `os.DirEntry` ne sont pas censées être stockées dans des structures de données à longue durée de vie; si vous savez que les métadonnées du fichier ont changé ou si un certain temps s'est écoulé depuis l'appel à `scandir()`, appelez `os.stat(entry.path)` pour mettre à jour ces informations.

Puisque les méthodes de `os.DirEntry` peuvent réaliser des appels système, elles peuvent aussi lever des `OSError`. Si vous avez besoin d'un contrôle fin des erreurs, vous pouvez attraper les `OSError` en appelant les méthodes de `os.DirEntry` et les traiter comme il vous semble.

Pour être directement utilisable comme un *path-like object*, `os.DirEntry` implémente l'interface *PathLike*.

Les attributs et méthodes des instances de `os.DirEntry` sont les suivants :

name

Le nom de fichier de base de l'entrée, relatif à l'argument `path` de `scandir()`.

L'attribut `name` sera de type `bytes` si l'argument `path` de `scandir()` est de type `bytes`, sinon il sera de type `str`. Utilisez `fsdecode()` pour décoder des noms de fichiers de types *byte*.

path

Le nom entier de l'entrée : équivalent à `os.path.join(scandir_path, entry.name)` où `scandir_path` est l'argument `path` de `scandir()`. Le chemin est absolu uniquement si l'argument `path` de `scandir()` était absolu.

L'attribut `path` sera de type `bytes` si l'argument `path` de la fonction `scandir()` est de type `bytes`, sinon il sera de type `str`. Utilisez `fsdecode()` pour décoder des noms de fichiers de type *bytes*.

inode()

Renvoie le numéro d'*inode* de l'entrée.

Le résultat est mis en cache dans l'objet `os.DirEntry`. Utilisez `os.stat(entry.path, follow_symlinks=False).st_ino` pour obtenir l'information à jour.

Au premier appel non mis en cache, un appel système est requis sur Windows, mais pas sur Unix.

is_dir (*, `follow_symlinks=True`)

Renvoie `True` si cette entrée est un répertoire ou un lien symbolique pointant vers un répertoire; renvoie `False` si l'entrée est (ou pointe vers) un autre type de fichier, ou s'il n'existe plus.

Si `follow_symlinks` vaut `False`, renvoie `True` uniquement si l'entrée est un répertoire (sans suivre les liens symboliques); renvoie `False` si l'entrée est n'importe quel autre type de fichier ou s'il n'existe plus.

Le résultat est mis en cache dans l'objet `os.DirEntry`, avec un cache séparé pour les valeurs `True` ou `False` de `follow_symlinks`. Appelez `os.stat()` avec `stat.S_ISDIR()` pour obtenir l'information à jour.

Au premier appel non mis en cache, aucun appel système n'est requis dans la plupart du temps. Spécifiquement, sans les liens symboliques, ni Windows, ni Unix ne requiert l'appel système, sauf sur certains systèmes de fichiers sur Unix, comme les système de fichiers de réseau qui renvoient `dirent.d_type ==`

`DT_UNKNOWN`. Si l'entrée est un lien symbolique, un appel système sera requis pour suivre le lien symbolique, à moins que `follow_symlinks` vaille `False`.

Cette méthode peut lever une `OSError` tout comme une `PermissionError`, mais `FileNotFoundError` est attrapé et pas levé.

`is_file` (*, `follow_symlinks=True`)

Renvoie `True` si l'entrée est un fichier ou un lien symbolique pointant vers un fichier, renvoie `False` si l'entrée pointe est (ou pointe sur) sur un dossier ou sur un répertoire ou autre entrée non-fichier, ou s'il n'existe plus.

Si `follow_symlinks` vaut `False`, renvoie `True` uniquement si cette entrée est un fichier (sans suivre les liens symboliques). Renvoie `False` si l'entrée est un répertoire ou une autre entrée non-fichier, ou s'il n'existe plus.

Le résultat est mis en cache dans l'objet `os.DirEntry`. La mise en cache, les appels système réalisés, et les exceptions levées sont les mêmes que pour `is_dir()`.

`is_symlink` ()

Renvoie `True` si l'entrée est un lien symbolique (même cassé). Renvoie `False` si l'entrée pointe vers un répertoire ou tout autre type de fichier, ou s'il n'existe plus.

Le résultat est mis en cache dans l'objet `os.DirEntry`. Appelez `os.path.islink()` pour obtenir l'information à jour.

Au premier appel non mis en cache, aucun appel système n'est requis. Spécifiquement, ni Windows ni Unix ne requiert d'appel système, excepté sur certains systèmes de fichiers Unix qui renvoient `dirent.d_type == DT_UNKNOWN`.

Cette méthode peut lever une `OSError` tout comme une `PermissionError`, mais `FileNotFoundError` est attrapé et pas levé.

`stat` (*, `follow_symlinks=True`)

Renvoie un objet de type `stat.result` pour cette entrée. Cette méthode suit les liens symboliques par défaut. Pour avoir les statistiques sur un lien symbolique, ajouter l'argument `follow_symlinks=False`.

Sur Unix, cette méthode requiert toujours un appel système. Sur Windows, cela requiert uniquement un appel système si `follow_symlinks` vaut `True` et l'entrée n'est pas un lien symbolique.

Sur Windows, les attributs `st_ino`, `st_dev` et `st_nlink` de la classe `stat_result` sont toujours définis à 0. Appelez la fonction `os.stat()` pour avoir ces attributs.

Le résultat est mis en cache dans l'objet `os.DirEntry`, avec un cache séparé pour les valeurs `True` ou `False` de `follow_symlinks`. Appelez `os.stat()` pour obtenir l'information à jour.

Notez qu'il y a une correspondance entre différents attributs et méthodes de `os.DirEntry` et `pathlib.Path`. En particulier, l'attribut `name` a la même signification, ainsi que les méthodes `is_dir()`, `is_file()`, `is_symlink()` et `stat()`.

Nouveau dans la version 3.5.

Modifié dans la version 3.6 : Prise en charge de l'interface `PathLike`. Ajout du support des chemins `bytes` sous Windows.

`os.stat` (`path`, *, `dir_fd=None`, `follow_symlinks=True`)

Récupère le statut d'un fichier ou d'un descripteur de fichier. Réalise l'équivalent d'un appel système `stat()` sur le chemin donné. `path` peut être exprimé comme une chaîne de caractères ou d'octets – directement ou indirectement à travers une interface `PathLike` – ou comme un descripteur de fichier ouvert. Renvoie un objet `stat_result`. Cette fonction suit normalement les liens symboliques. Pour récupérer les informations d'un lien symbolique, ajoutez l'argument `follow_symlinks=False` ou utilisez la fonction `lstat()`.

Cette fonction peut supporter la *spécification d'un descripteur de fichier* et le *non-suivi des liens symboliques*.

Exemple :

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
```

(suite sur la page suivante)

(suite de la page précédente)

```
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

Voir aussi :

les fonctions `fstat()` et `lstat()`.

Nouveau dans la version 3.3 : Les arguments `dir_fd` et `follow_symlinks` ont été ajoutés, spécification d'un descripteur de fichier à la place d'un chemin ajoutée également.

Modifié dans la version 3.6 : Accepte un *path-like object*.

class os.stat_result

Objet dont les attributs correspondent globalement aux membres de la structure `stat()`. Utilisé pour le résultat des fonctions `os.stat()`, `os.fstat()`, et `os.lstat()`.

Attributs :

st_mode

Mode du fichier : type du fichier et bits de mode du fichier (permissions).

st_ino

Dépendant de la plateforme, mais lorsqu'il ne vaut pas zéro il identifie de manière unique le fichier pour une certaine valeur de `st_dev`. Typiquement :

- le numéro d'*inode* sur Unix,
- l'*index de fichier* sur Windows

st_dev

Identifiant du périphérique sur lequel ce fichier se trouve.

st_nlink

Nombre de liens matériels.

st_uid

Identifiant d'utilisateur du propriétaire du fichier.

st_gid

Identifiant de groupe du propriétaire du fichier.

st_size

Taille du fichier en *bytes* si c'est un fichier normal ou un lien symbolique. La taille d'un lien symbolique est la longueur du nom de chemin qu'il contient sans le byte nul final.

Horodatages :

st_atime

Moment de l'accès le plus récent, exprimé en secondes.

st_mtime

Moment de la modification de contenu la plus récente, exprimé en secondes.

st_ctime

Dépendant de la plate-forme :

- le moment du changement de méta-données le plus récent sur Unix,
- le moment de création sur Windows, exprimé en secondes.

st_atime_ns

Moment de l'accès le plus récent, exprimé en nanosecondes, par un entier.

st_mtime_ns

Moment de la modification de contenu la plus récente, exprimé en nanosecondes, par un entier.

st_ctime_ns

Dépendant de la plate-forme :

- le moment du changement de méta-données le plus récent sur Unix,
- le moment de création sur Windows, exprimé en nanosecondes, par un entier.

Voir aussi la fonction `stat_float_times()`.

Note : La signification et la précision exacte des attributs `st_atime`, `st_mtime`, et `st_ctime` dépendent du système d'exploitation et du système de fichier. Par exemple sur les systèmes Windows qui utilisent un système de fichier FAT ou FAT32, `st_mtime` a une précision de 2 secondes, et `st_atime` a une précision de 1 jour. Regardez la documentation de votre système d'exploitation pour plus de détails.

De manière similaire, bien que `st_atime_ns`, `st_mtime_ns`, et `st_ctime_ns` soient toujours exprimés en nanosecondes, beaucoup de systèmes ne fournissent pas une précision à la nanoseconde près. Sur les systèmes qui fournissent une telle précision, l'objet à virgule flottante utilisé pour stocker `st_atime`, `st_mtime`, et `st_ctime` ne peut pas le contenir en entier, et donc sera légèrement inexact. Si vous avez besoin d'horodatages exacts, vous devriez toujours utiliser `st_atime_ns`, `st_mtime_ns`, et `st_ctime_ns`.

Sur certains systèmes Unix (tels que Linux), les attributs suivants peuvent également être disponibles :

`st_blocks`

Nombre de blocs de 512 *bytes* alloués pour le fichier. Cette valeur peut être inférieure à `st_size/512` quand le fichier a des trous.

`st_blksize`

Taille de bloc « préférée » pour des E/S efficaces avec le système de fichiers. Écrire dans un fichier avec des blocs plus petits peut causer des modifications (lecture-écriture-réécriture) inefficaces.

`st_rdev`

Type de périphérique si l'*inode* représente un périphérique.

`st_flags`

Marqueurs définis par l'utilisateur pour le fichier.

Sur d'autres systèmes Unix (tels que FreeBSD), les attributs suivants peuvent être disponibles (mais peuvent être complétés uniquement lorsque le super-utilisateur *root* tente de les utiliser) :

`st_gen`

Nombre de génération de fichier.

`st_birthtime`

Moment de la création du fichier.

Sur les systèmes Mac OS, les attributs suivants peuvent également être disponibles :

`st_rsize`

Taille réelle du fichier.

`st_creator`

Créateur du fichier.

`st_type`

Type du fichier.

Sur les systèmes Windows, les attributs suivants sont également disponibles :

`st_file_attributes`

Attributs de fichiers Windows : membre `dwFileAttributes` de la structure `BY_HANDLE_FILE_INFORMATION` renvoyée par `GetFileInformationByHandle()`. Soit les constantes `FILE_ATTRIBUTE_*` du module `stat`.

Le module standard `stat` définit des fonctions et des constantes qui sont utiles pour l'extraction d'informations d'une structure `stat`. (Sur Windows, certains éléments sont remplis avec des valeurs factices.)

Pour des raisons de rétro-compatibilité, une instance du `stat_result` est également accessible comme un tuple d'au moins 10 valeurs entières donnant les membres les plus importants (et portables) de la structure `stat`, dans l'ordre : `st_mode`, `st_ino`, `st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime`, `st_ctime`. Plus d'éléments peuvent être ajoutés à la fin par certaines implémentations. Pour une compatibilité avec les anciennes versions de Python, accéder à un élément de type `stat_result` comme un tuple donne toujours des entiers.

Nouveau dans la version 3.3 : Les attributs `st_atime_ns`, `st_mtime_ns`, et `st_ctime_ns` ont été ajoutés.

Nouveau dans la version 3.5 : L'attribut `st_file_attributes` a été ajouté sur Windows.

Modifié dans la version 3.5 : Windows renvoie maintenant l'index du fichier dans l'attribut `st_ino`, lorsqu'il est disponible.

os.stat_float_times([newvalue])

Détermine si `stat_result` représente les horodatages par des objets à virgule flottante. Si `newvalue` vaut `True`, les appels à `stat()` qui suivront renverront des flottants. Si `newvalue` vaut `False`, les appels qui suivront renverront des entiers. Si `newvalue` est omise, la valeur actuelle est renvoyée.

Pour des raisons de compatibilité avec les anciennes versions de Python, accéder un objet de type `stat_result` renvoie toujours des entiers.

Python renvoie des valeurs flottantes par défaut maintenant. Les applications qui ne fonctionnent pas correctement avec des horodatages flottants peuvent utiliser cette fonction pour restaurer l'ancien comportement.

La précision des horodatages (qui est la plus petite fraction possible) dépend du système. Certains systèmes supportent uniquement une précision à la seconde ; sur ces systèmes, la fraction sera toujours zéro.

Il est recommandé que cette option ne soit changée qu'au lancement du programme dans le module `__main__`. Les bibliothèques ne devraient jamais changer cette option. Si une application utilise une bibliothèque qui fonctionne mal avec les horodatages en nombres flottants, cette application devrait temporairement retirer cette possibilité jusqu'à ce que la bibliothèque ait été corrigée.

Obsolète depuis la version 3.3.

os.statvfs(path)

Exécute un appel système `statvfs()` sur le chemin donné par `path`. La valeur de retour est un objet dont les attributs décrivent le système de fichier sur le chemin donné, et correspondent aux membres de la structure `statvfs` nommés : `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`.

Deux constantes de module sont définies pour le champ-de-bits de l'attribut `f_flag` : si `SR_RDONLY` est activé, le système de fichiers est monté en lecture-seule, et si `ST_NOSUID` est activé, la sémantique des bits de `setuid` / `getuid` est désactivée ou non gérée.

Des constantes de module supplémentaires sont définies pour les systèmes basés sur GNU/glibc. Ces constantes sont `ST_NODEV` (interdit l'accès aux fichiers spéciaux du périphérique), `ST_NOEXEC` (interdit l'exécution de programmes), `ST_SYNCHRONOUS` (les écritures sont synchronisées en une fois), `ST_MANDLOCK` (permet les verrous impératifs sur un système de fichiers), `ST_WRITE` (écrit sur les fichiers/répertoires/liens symboliques), `ST_APPEND` (fichiers en ajout-seul), `ST_IMMUTABLE` (fichiers immuables), `ST_NOATIME` (ne met pas à jour les moments d'accès), `ST_NODIRATIME` (ne met pas à jour les moments d'accès aux répertoires), `ST_REALTIME` (Met `atime` à jour relativement à `mtime` / `ctime`).

Cette fonction prend en charge *la spécification d'un descripteur de fichier*.

Disponibilité : Unix.

Modifié dans la version 3.2 : Ajout des constantes `ST_RDONLY` et `ST_NOSUID`.

Nouveau dans la version 3.3 : Prise en charge de la spécification d'un descripteur de répertoire pour `path` ajouté.

Modifié dans la version 3.4 : Ajout des constantes `ST_NODEV`, `ST_NOEXEC`, `ST_SYNCHRONOUS`, `ST_MANDLOCK`, `ST_WRITE`, `ST_APPEND`, `ST_IMMUTABLE`, `ST_NOATIME`, `ST_NODIRATIME`, et `ST_RELATIME`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.supports_dir_fd

Un objet de type `Set` indiquant quels fonctions du module `os` permettent l'utilisation du paramètre `dir_fd`. Des plate-formes différentes fournissent des fonctionnalités différentes, et une option qui peut fonctionner sur l'une peut ne pas être gérée sur une autre. Pour des raisons de cohérence, les fonctions qui gèrent `dir_fd` permettent toujours de spécifier le paramètre, mais lèvent une exception si la fonctionnalité n'est pas réellement accessible.

Pour vérifier si une fonction en particulier permet de l'utilisation de son paramètre `dir_fd`, utilisez l'opérateur `in` sur `supports_dir_fd`. Par exemple, l'expression détermine si le paramètre `dir_fd` de la fonction `os.stat()` est disponible :

```
os.stat in os.supports_dir_fd
```

Actuellement, le paramètre `dir_fd` ne fonctionne que sur les plate-formes Unix. Il ne fonctionne jamais sur Windows.

Nouveau dans la version 3.3.

`os.supports_effective_ids`

Un objet de type `collections.abc.Set` indiquant quelles fonction du module `ps` permettent l'utilisation du paramètre `effective_ids` pour `os.access()`. Si la plate-forme le gère, la collection contiendra `os.access()`, sinon elle sera vide.

Pour vérifier si vous pouvez utiliser le paramètre `effective_ids` pour `os.access()`, utilisez l'opérateur `in` sur `supports_effective_ids`, comme tel :

```
os.access in os.supports_effective_ids
```

Actuellement, `effective_ids` ne fonctionne que sur les plate-formes Unix, ça ne fonctionne pas sur Windows.

Nouveau dans la version 3.3.

`os.supports_fd`

Un objet de type `Set` indiquant quelles fonctions du module `os` permettent de spécifier le paramètre de chemin `path` par un descripteur de fichier ouvert. Différentes plate-formes fournissent différentes fonctionnalités, et une option qui peut fonctionner sur l'une peut ne pas être gérée sur une autre. Pour des raisons de cohérence, les fonctions qui gèrent `fd` permettent toujours de spécifier le paramètre, mais elles lèveront une exception si la fonctionnalité n'est pas réellement disponible.

Pour vérifier si une fonction en particulier permet de spécifier un descripteur de fichier ouvert pour son paramètre `path`, utilisez l'opérateur `in` sur `supports_fd`. Par exemple, cette expression détermine si `os.chdir()` accepte un descripteur de fichier ouvert quand appelée sur votre plate-forme actuelle :

```
os.chdir in os.supports_fd
```

Nouveau dans la version 3.3.

`os.supports_follow_symlinks`

Un objet de type `collections.abc.Set` indiquant quelles fonctions du module `os` permettent d'utiliser leur paramètre `follow_symlinks`. Différentes plate-formes fournissent des fonctionnalités différentes, et une option qui fonctionne sur l'une peut ne pas fonctionner sur une autre. Pour des raisons de cohérence, les fonctions qui gèrent `follow_symlinks` permettent toujours de spécifier le paramètre, mais lèvent une exception si la fonctionnalité n'est pas réellement disponible.

Pour vérifier si une fonction en particulier permet l'utilisation du paramètre `follow_symlinks`, utilisez l'opérateur `in` sur `supports_follow_symlinks`. Par exemple, cette expression détermine si le paramètre `follow_symlink` de `os.stat()` est disponible :

```
os.stat in os.supports_follow_symlinks
```

Nouveau dans la version 3.3.

`os.symlink(src, dst, target_is_directory=False, *, dir_fd=None)`

Crée un lien symbolique pointant vers `src` et appelé `dst`.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if `target_is_directory` is `True` or a file symlink (the default) otherwise. On non-Windows platforms, `target_is_directory` is ignored.

Introduction de la prise en charge des liens symboliques dans Windows 6.0 (Vista). `symlink()` lèvera une exception `NotImplementedError` sur les versions de Windows inférieures à 6.0.

Cette fonction prend en charge *des chemins relatifs à des descripteurs de répertoires*.

Note : Sur Windows, le `SeCreateSymbolicLinkPrivilege` est requis pour créer des liens symboliques avec succès. Ce privilège n'est pas typiquement garanti aux utilisateurs réguliers mais est disponible aux comptes qui peuvent escalader les privilèges jusqu'au niveau administrateur. Tant obtenir le privilège que lancer votre application en administrateur sont des moyens de créer des liens symboliques avec succès.

`OSError` est levée quand la fonction est appelée par un utilisateur sans privilèges.

Disponibilité : Unix, Windows.

Modifié dans la version 3.2 : Prise en charge des les liens symboliques sur Windows 6.0 (Vista).

Nouveau dans la version 3.3 : Ajout de l'argument *dir_fd* et maintenant, permission de *target_is_directory* sur les plate-formes non Windows.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *src* et *dst*.

os.sync()

Force l'écriture de tout sur le disque.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.truncate(path, length)

Tronque le fichier correspondant à *path*, afin qu'il soit au maximum long de *length* bytes.

Cette fonction prend en charge *la spécification d'un descripteur de fichier*.

Disponibilité : Unix, Windows.

Nouveau dans la version 3.3.

Modifié dans la version 3.5 : Prise en charge de Windows

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.unlink(path, *, dir_fd=None)

Supprime (retire) le fichier *path*. Cette fonction est sémantiquement identique à *remove()*. Le nom *unlink* est un nom Unix traditionnel. Veuillez voir la documentation de *remove()* pour plus d'informations.

Nouveau dans la version 3.3 : Le paramètre *dir_fd*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.utime(path, times=None, *[ns], dir_fd=None, follow_symlinks=True)

Voir les derniers moments d'accès et de modification du fichier spécifiés par *path*.

La fonction *utime()* prend deux paramètres optionnels, *times* et *ns*. Ils spécifient le temps mis pour *path* et est utilisé comme suit :

- Si *ns* est spécifié, ce doit être un couple de la forme (*atime_ns*, *mtime_ns*) où chaque membre est un entier qui exprime des nanosecondes.
- Si *times* ne vaut pas *None*, ce doit être un couple de la forme (*atime*, *mtime*) où chaque membre est un entier ou une expression à virgule flottante.
- Si *times* vaut *None*, et *ns* est non-spécifié. C'est équivalent à spécifier *ns* = (*atime_ns*, *mtime_ns*) où les deux moments sont le moment actuel.

Il est erroné de spécifier des tuples pour *times* et *ns* à la fois.

Le fait qu'un répertoire puisse être donné pour *path* dépend du fait que le système d'exploitation implémente les répertoires comme des fichiers (par exemple, Windows ne le fait pas). Notez que l'instant exact que vous définissez ici peut ne pas être renvoyé lors d'un futur appel à *stat()*, selon la précision avec laquelle votre système d'exploitation mémorise les moments d'accès et de modification ; voir *stat()*. Le meilleur moyen de préserver des moments exacts est d'utiliser les champs *st_atime_ns* et *st_mtime_ns* de l'objet résultat de la fonction *os.stat()* avec le paramètre *ns* valant *utime*.

Cette fonction prend en charge *la spécification d'un descripteur de fichier, les chemins relatifs à des descripteurs de répertoires, et le non-suivi des liens symboliques*.

Nouveau dans la version 3.3 : Prise en charge de la spécification d'un descripteur de fichier pour *path* et les paramètres *dir_fd*, *follow_symlinks*, et *ns* ajoutés.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.walk(top, topdown=True, onerror=None, followlinks=False)

Génère les noms de fichier dans un arbre de répertoire en parcourant l'arbre soit de bas-en-haut, soit de haut-en-bas. Pour chaque répertoire dans l'arbre enraciné en le répertoire *top* (incluant ledit répertoire *top*), fournit un triplet (*dirpath*, *dirnames*, *filenames*).

dirpath est une chaîne de caractères contenant le chemin vers le répertoire. *dirnames* est une liste de noms de sous-répertoires dans *dirpath* (en excluant '.' et '..'). *filenames* est une liste de fichiers (non-répertoires) dans

dirpath. Notez que les noms dans la liste ne contiennent aucune composante de chemin. Pour récupérer le chemin complet (qui commence à *top*) vers un répertoire dans *dirpath*, faites `os.path.join(dirpath, name)`.

Si l'argument optionnel *topdown* vaut `True` ou n'est pas spécifié, le triplet pour un répertoire est généré avant les triplets de tous ses sous-répertoires (les répertoires sont générés de haut-en-bas). Si *topdown* vaut `False`, le triplet pour un répertoire est généré après les triplets de tous ses sous-répertoires (les répertoires sont générés de bas-en-haut). Peu importe la valeur de *topdown*, la liste des sous-répertoires est récupérée avant que les tuples pour les répertoires et ses sous-répertoires ne soient générés.

Quand *topdown* vaut `True`, l'appelant peut modifier la liste *dirnames* en place (par exemple en utilisant `del` ou l'assignation par *slicing* (par tranches)) et *walk()* ne fera sa récursion que dans les sous-répertoires dont le nom reste dans *dirnames*; cela peut être utilisé pour élaguer la recherche, imposer un ordre particulier de visite, ou encore pour informer *walk()* des répertoires créés par l'appelant ou renommés avant qu'il quitte *walk()* à nouveau. Modifier *dirnames* quand *topdown* vaut `False` n'a aucun effet sur le comportement du parcours parce qu'en mode bas-en-haut, les répertoires dans *dirnames* sont générés avant que *dirpath* ne soit lui-même généré.

Par défaut, les erreurs d'un appel à *scandir()* sont ignorées. Si l'argument optionnel *onerror* est spécifié, il doit être une fonction qui sera appelée avec un seul argument, une instance de *OSError*. Elle peut rapporter l'erreur et continuer le parcours, ou lever l'exception pour avorter le parcours. Notez que le nom de fichier est disponible dans l'attribut *filename* de l'objet exception.

Par défaut, *walk()* ne parcourra pas les liens symboliques qui mènent à un répertoire. Définissez *followlinks* avec `True` pour visiter les répertoires pointés par des liens symboliques sur les systèmes qui le gère.

Note : Soyez au courant que définir *followlinks* avec `True` peut mener à une récursion infinie si un lien pointe vers un répertoire parent de lui-même. *walk()* ne garde pas de trace des répertoires qu'il a déjà visités.

Note : Si vous passez un chemin relatif, ne changez pas le répertoire de travail actuel entre deux exécutions de *walk()*. *walk()* ne change jamais le répertoire actuel, et suppose que l'appelant ne le fait pas non plus.

Cet exemple affiche le nombre de bytes pris par des fichiers non-répertoires dans chaque répertoire à partir du répertoire de départ, si ce n'est qu'il ne cherche pas après un sous-répertoire CSV :

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS') # don't visit CVS directories
```

Dans l'exemple suivant (simple implémentation d'un *shutil.rmtree()*), parcourir l'arbre de bas-en-haut est essentiel : *rmdir()* ne permet pas de supprimer un répertoire avant qu'un ne soit vide :

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION: This is dangerous! For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

Modifié dans la version 3.5 : Cette fonction appelle maintenant *os.scandir()* au lieu de *os.listdir()*, ce qui la rend plus rapide en réduisant le nombre d'appels à *os.stat()*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.fwalk` (*top*='.', *topdown*=True, *onerror*=None, *, *follow_symlinks*=False, *dir_fd*=None)

Se comporte exactement comme `walk()`, si ce n'est qu'il fournit un quadruplet (*dirpath*, *dirnames*, *filenames*, *dirfd*), et gère *dir_fd*.

dirpath, *dirnames* et *filenames* sont identiques à la sortie de `walk()` et *dirfd* est un descripteur de fichier faisant référence au répertoire *dirpath*.

Cette fonction prend toujours en charge *les chemins relatifs à des descripteurs de fichiers* et *le non-suivi des liens symboliques*. Notez cependant qu'à l'inverse des autres fonctions, la valeur par défaut de *follow_symlinks* pour `walk()` est False.

Note : Puisque `fwalk()` fournit des descripteurs de fichiers, ils ne sont valides que jusque la prochaine itération. Donc vous devriez les dupliquer (par exemple avec `dup()`) si vous désirez les garder plus longtemps.

Cet exemple affiche le nombre de bytes pris par des fichiers non-répertoires dans chaque répertoire à partir du répertoire de départ, si ce n'est qu'il ne cherche pas après un sous-répertoire CSV :

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end="")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS') # don't visit CVS directories
```

Dans le prochain exemple, parcourir l'arbre de bas-en-haut est essentiel : `rmdir()` ne permet pas de supprimer un répertoire avant qu'il ne soit vide :

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION: This is dangerous! For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

Disponibilité : Unix.

Nouveau dans la version 3.3.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Attributs étendus pour Linux

Nouveau dans la version 3.3.

Toutes ces fonctions ne sont disponibles que sur Linux.

`os.getxattr` (*path*, *attribute*, *, *follow_symlinks*=True)

Renvoie la valeur de l'attribut étendu *attribute* du système de fichiers pour le chemin *path*. *attribute* peut être une chaîne de caractères ou d'octets (directement ou indirectement à travers une interface *PathLike*). Si c'est une chaîne de caractères, elle est encodée avec l'encodage du système de fichiers.

Cette fonction peut supporter la *spécification d'un descripteur de fichier* et *le non-suivi des liens symboliques*.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *path* et *attribute*.

os.**listxattr** (*path=None*, *, *follow_symlinks=True*)

Renvoie une liste d'attributs du système de fichiers étendu pour *path*. Les attributs dans la liste sont représentés par des chaînes de caractères et sont décodés avec l'encodage du système de fichier. Si *path* vaut *None*, *listxattr()* examinera le répertoire actuel.

Cette fonction peut supporter la *spécification d'un descripteur de fichier* et le *non-suivi des liens symboliques*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.**removexattr** (*path*, *attribute*, *, *follow_symlinks=True*)

Supprime l'attribut étendu *attribute* du système de fichier pour le chemin *path*. *attribute* devrait être une chaîne de caractères ou d'octets (directement ou indirectement à travers une interface *PathLike*). Si c'est une chaîne de caractères, elle est encodée avec l'encodage du système de fichiers.

Cette fonction peut supporter la *spécification d'un descripteur de fichier* et le *non-suivi des liens symboliques*.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *path* et *attribute*.

os.**setxattr** (*path*, *attribute*, *value*, *flags=0*, *, *follow_symlinks=True*)

Règle l'attribut étendu *attribute* du système de fichier pour le chemin *path* à *value*. *attribute* doit être une chaîne de caractères ou d'octets sans caractères nuls (directement ou indirectement à travers une interface *PathLike*). Si c'est une chaîne de caractères, elle est encodée avec l'encodage du système de fichiers. *flags* peut être *XATTR_REPLACE* ou *XATTR_CREATE*. Si *XATTR_REPLACE* est donné et que l'attribut n'existe pas, *EEXIST* sera levée. Si *XATTR_CREATE* est donné et que l'attribut existe déjà, l'attribut ne sera pas créé et *ENODATA* sera levée.

Cette fonction peut supporter la *spécification d'un descripteur de fichier* et le *non-suivi des liens symboliques*.

Note : Un bug des versions inférieures à 2.6.39 du noyau Linux faisait que les marqueurs de *flags* étaient ignorés sur certains systèmes.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *path* et *attribute*.

os.**XATTR_SIZE_MAX**

La taille maximum que peut faire la valeur d'un attribut étendu. Actuellement, c'est 64 KiB sur Lniux.

os.**XATTR_CREATE**

C'est une valeur possible pour l'argument *flags* de *setxattr()*. Elle indique que l'opération doit créer un attribut.

os.**XATTR_REPLACE**

C'est une valeur possible pour l'argument *flags* de *setxattr()*. Elle indique que l'opération doit remplacer un attribut existant.

16.1.6 Gestion des processus

Ces fonctions peuvent être utilisées pour créer et gérer des processus.

Les variantes des fonctions *exec** prennent une liste d'arguments pour le nouveau programme chargé dans le processus. Dans tous les cas, le premier de ces arguments est passé au nouveau programme comme son propre nom plutôt que comme un argument qu'un utilisateur peut avoir tapé en ligne de commande. Pour les programmeurs C, c'est l'argument *argv[0]* qui est passé à la fonction *main()* du programme. Par exemple, *os.execv('/bin/echo/', ['foo', 'bar'])* affichera uniquement *bar* sur la sortie standard ; *foo* semblera être ignoré.

os.**abort** ()

Génère un signal *SIGABRT* au processus actuel. Sur Linux, le comportement par défaut est de produire un vidage système (*Core Dump*) ; sur Windows, le processus renvoie immédiatement un code d'erreur 3. Attention : appeler cette fonction n'appellera pas le gestionnaire de signal Python enregistré par *SIGABRT* à l'aide de *signal.signal()*.

os.**execl** (*path*, *arg0*, *arg1*, ...)

os.**execle** (*path*, *arg0*, *arg1*, ..., *env*)

os.**execlp** (*file*, *arg0*, *arg1*, ...)

```

os.execlpe (file, arg0, arg1, ..., env)
os.execv (path, args)
os.execve (path, args, env)
os.execvp (file, args)
os.execvpe (file, args, env)

```

Ces fonctions exécutent toutes un nouveau programme, remplaçant le processus actuel, elles ne renvoient pas. Sur Unix, le nouvel exécutable est chargé dans le processus actuel, et aura le même identifiant de processus (PID) que l'appelant. Les erreurs seront reportées par des exceptions `OSError`.

Le processus actuel est remplacé immédiatement. Les fichiers objets et descripteurs de fichiers ne sont pas purgés, donc s'il est possible que des données aient été mises en tampon pour ces fichiers, vous devriez les purger manuellement en utilisant `sys.stdout.flush()` ou `os.fsync()` avant d'appeler une fonction `exec*`.

Les variantes « l » et « v » des fonctions `exec*` diffèrent sur la manière de passer les arguments de ligne de commande. Les variantes « l » sont probablement les plus simples à utiliser si le nombre de paramètres est fixé lors de l'écriture du code. Les paramètres individuels deviennent alors des paramètres additionnels aux fonctions `exec*()`. Les variantes « v » sont préférables quand le nombre de paramètres est variable et qu'ils sont passés dans une liste ou un *tuple* dans le paramètre `args`. Dans tous les cas, les arguments aux processus fils devraient commencer avec le nom de la commande à lancer, mais ce n'est pas obligatoire.

Les variantes qui incluent un « p » vers la fin (`execlp()`, `execlpe()`, `execvp()`, et `execvpe()`) utiliseront la variable d'environnement `PATH` pour localiser le programme `file`. Quand l'environnement est remplacé (en utilisant une des variantes `exec*e`, discutées dans le paragraphe suivant), le nouvel environnement est utilisé comme source de la variable d'environnement `PATH`. Les autres variantes `execl()`, `execle()`, `execv()`, et `execve()` n'utiliseront pas la variable d'environnement `PATH` pour localiser l'exécutable. `path` doit contenir un chemin absolue ou relatif approprié.

Pour les fonctions `execle()`, `execlpe()`, `execve()`, et `execvpe()` (notez qu'elle finissent toutes par « e »), le paramètre `env` doit être un *mapping* qui est utilisé pour définir les variables d'environnement du nouveau processus (celles-ci sont utilisées à la place de l'environnement du nouveau processus). Les fonctions `execl()`, `execlp()`, `execv()`, et `execvp()` causent toutes un héritage de l'environnement du processus actuel par le processus fils.

Pour `execve()`, sur certaines plate-formes, `path` peut également être spécifié par un descripteur de fichier ouvert. Cette fonctionnalité peut ne pas être gérée sur votre plate-forme. Vous pouvez vérifier si c'est disponible en utilisant `os._supports_fd`. Si c'est indisponible, l'utiliser lèvera une `NotImplementedError`.

Disponibilité : Unix, Windows.

Nouveau dans la version 3.3 : Prise en charge de la spécification d'un descripteur de fichier ouvert pour `path` pour `execve()`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

```

os._exit (n)

```

Quitte le processus avec le statut `n`, sans appeler les gestionnaires de nettoyage, sans purger les tampons des fichiers, etc.

Note : La méthode standard pour quitter est `sys.exit(n)`. `_exit()` devrait normalement être utilisé uniquement par le processus fils après un `fork()`.

Les codes de sortie suivants sont définis et peuvent être utilisés avec `_exit()`, mais ils ne sont pas nécessaires. Ils sont typiquement utilisés pour les programmes systèmes écrits en Python, comme un programme de gestion de l'exécution des commandes d'un serveur de mails.

Note : Certaines de ces valeurs peuvent ne pas être disponibles sur toutes les plate-formes Unix étant donné qu'il en existe des variations. Ces constantes sont définies là où elles sont définies par la plate-forme sous-jacente.

```

os.EX_OK

```

Code de sortie signifiant qu'aucune erreur n'est arrivée.

Disponibilité : Unix.

OS.**EX_USAGE**

Code de sortie signifiant que les commandes n'ont pas été utilisées correctement, comme quand le mauvais nombre d'arguments a été donné.

Disponibilité : Unix.

OS.**EX_DATAERR**

Code de sortie signifiant que les données en entrées étaient incorrectes.

Disponibilité : Unix.

OS.**EX_NOINPUT**

Code de sortie signifiant qu'un des fichiers d'entrée n'existe pas ou n'est pas lisible.

Disponibilité : Unix.

OS.**EX_NOUSER**

Code de sortie signifiant qu'un utilisateur spécifié n'existe pas.

Disponibilité : Unix.

OS.**EX_NOHOST**

Code de sortie signifiant qu'un hôte spécifié n'existe pas.

Disponibilité : Unix.

OS.**EX_UNAVAILABLE**

Code de sortie signifiant qu'un service requis n'est pas disponible.

Disponibilité : Unix.

OS.**EX_SOFTWARE**

Code de sortie signifiant qu'une erreur interne d'un programme a été détectée.

Disponibilité : Unix.

OS.**EX_OSERR**

Code de sortie signifiant qu'une erreur du système d'exploitation a été détectée, comme l'incapacité à réaliser un *fork* ou à créer un tuyau (*pipe*).

Disponibilité : Unix.

OS.**EX_OSFILE**

Code de sortie signifiant qu'un fichier n'existe pas, n'a pas pu être ouvert, ou avait une autre erreur.

Disponibilité : Unix.

OS.**EX_CANTCREAT**

Code de sortie signifiant qu'un fichier spécifié par l'utilisateur n'a pas pu être créé.

Disponibilité : Unix.

OS.**EX_IOERR**

Code de sortie signifiant qu'une erreur est apparue pendant une E/S sur un fichier.

Disponibilité : Unix.

OS.**EX_TEMPFAIL**

Code de sortie signifiant qu'un échec temporaire est apparu. Cela indique quelque chose qui pourrait ne pas être une erreur, comme une connexion au réseau qui n'a pas pu être établie pendant une opération réessayable.

Disponibilité : Unix.

OS.**EX_PROTOCOL**

Code de sortie signifiant qu'un protocole d'échange est illégal, invalide, ou non-compris.

Disponibilité : Unix.

os.EX_NOPERM

Code de sortie signifiant qu'il manque certaines permissions pour réaliser une opération (mais n'est pas destiné aux problèmes de système de fichiers).

Disponibilité : Unix.

os.EX_CONFIG

Code de sortie signifiant qu'une erreur de configuration est apparue.

Disponibilité : Unix.

os.EX_NOTFOUND

Code de sortie signifiant quelque chose comme « une entrée n'a pas été trouvée ».

Disponibilité : Unix.

os.fork()

Fork un processus fils. Renvoie 0 dans le processus fils et le PID du processus fils dans le processus père. Si une erreur apparaît, une *OSError* est levée.

Notez que certaines plate-formes (dont FreeBSD <= 6.3 et Cygwin) ont des problèmes connus lors d'utilisation de *fork()* depuis un fil d'exécution.

Avertissement : Voir *ssl* pour les applications qui utilisent le module SSL avec *fork()*.

Disponibilité : Unix.

os.forkpty()

Fork un processus fils, en utilisant un nouveau pseudo-terminal comme terminal contrôlant le fils. Renvoie une paire (*pid*, *fd*) où *pid* vaut 0 dans le fils et le PID du processus fils dans le parent, et *fd* est le descripteur de fichier de la partie maître du pseudo-terminal. Pour une approche plus portable, utilisez le module *pty*. Si une erreur apparaît, une *OSError* est levée.

Disponibilité : certains dérivés Unix.

os.kill(pid, sig)

Envoie le signal *sig* au processus *pid*. Les constantes pour les signaux spécifiques à la plate-forme hôte sont définies dans le module *signal*.

Windows : les signaux *signal.CTRL_C_EVENT* et *signal.CTRL_BREAK_EVENT* sont des signaux spéciaux qui ne peuvent être envoyés qu'aux processus consoles qui partagent une console commune (par exemple, certains sous-processus). Toute autre valeur pour *sig* amènera le processus à être tué sans condition par l'API *TerminateProcess*, et le code de retour sera mis à *sig*. La version Windows de *kill()* prend en plus les identificateurs de processus à tuer.

Voir également *signal.thread_kill()*.

Nouveau dans la version 3.2 : Prise en charge de Windows.

os.killpg(pgid, sig)

Envoie le signal *sig* au groupe de processus *pgid*.

Disponibilité : Unix.

os.nice(increment)

Ajoute *increment* à la priorité du processus. Renvoie la nouvelle priorité.

Disponibilité : Unix.

os.plock(op)

Verrouille les segments du programme en mémoire. La valeur de *op* (définie dans *<sys/lock.h>*) détermine quels segments sont verrouillés.

Disponibilité : Unix.

os.popen(cmd, mode='r', buffering=-1)

Ouvre un tuyau vers ou depuis la commande *cmd*. La valeur de retour est un fichier objet ouvert relié au tuyau qui

peut être lu ou écrit selon la valeur de *mode* qui est 'r' (par défaut) ou 'w'. L'argument *buffering* a le même sens que l'argument correspondant de la fonction `open()`. L'objet fichier renvoyé écrit (ou lit) des chaînes de caractères et non de bytes.

La méthode `close` renvoie `None` si le sous-processus s'est terminé avec succès, ou le code de retour du sous-processus s'il y a eu une erreur. Sur les systèmes POSIX, si le code de retour est positif, il représente la valeur de retour du processus décalée d'un byte sur la gauche. Si le code de retour est négatif, le processus le processus s'est terminé avec le signal donné par la négation de la valeur de retour. (Par exemple, la valeur de retour pourrait être `-signal.SIGKILL` si le sous-processus a été tué). Sur les systèmes Windows, la valeur de retour contient le code de retour du processus fils dans un entier signé.

Ceci est implémenté en utilisant `subprocess.Popen`. Lisez la documentation de cette classe pour des méthodes plus puissantes pour gérer et communiquer avec des sous-processus.

```
os.spawnl(mode, path, ...)
os.spawnle(mode, path, ..., env)
os.spawnlp(mode, file, ...)
os.spawnlpe(mode, file, ..., env)
os.spawnv(mode, path, args)
os.spawnve(mode, path, args, env)
os.spawnvp(mode, file, args)
os.spawnvpe(mode, file, args, env)
```

Exécute le programme *path* dans un nouveau processus.

(Notez que le module `subprocess` fournit des outils plus puissants pour générer de nouveaux processus et récupérer leur valeur de retour. Il est préférable d'utiliser ce module que ces fonctions. Voyez surtout la section *Remplacer les fonctions plus anciennes par le module subprocess*.)

Si *mode* vaut `P_NOWAIT`, cette fonction renvoie le PID du nouveau processus, et si *mode* vaut `P_WAIT`, la fonction renvoie le code de sortie du processus s'il se termine normalement, ou `-signal` où *signal* est le signal qui a tué le processus. Sur Windows, le PID du processus sera en fait l'identificateur du processus (*process handle*) et peut donc être utilisé avec la fonction `waitpid()`.

Les variantes « l » et « v » des fonctions `spawn*` diffèrent sur la manière de passer les arguments de ligne de commande. Les variantes « l » sont probablement les plus simples à utiliser si le nombre de paramètres est fixé lors de l'écriture du code. Les paramètres individuels deviennent alors des paramètres additionnels aux fonctions `spawn*`. Les variantes « v » sont préférables quand le nombre de paramètres est variable et qu'ils sont passés dans une liste ou un *tuple* dans le paramètre *args*. Dans tous les cas, les arguments aux processus fils devraient commencer avec le nom de la commande à lancer, mais ce n'est pas obligatoire.

Les variantes qui incluent un « p » vers la fin (`spawnlp()`, `spawnlpe()`, `spawnvp()`, et `spawnvpe()`) utiliseront la variable d'environnement `PATH` pour localiser le programme *file*. Quand l'environnement est remplacé (en utilisant une des variantes `spawn*e`, discutées dans le paragraphe suivant), le nouvel environnement est utilisé comme source de la variable d'environnement `PATH`. Les autres variantes `spawnl()`, `spawnle()`, `spawnv()`, et `spawnve()` n'utiliseront pas la variable d'environnement `PATH` pour localiser l'exécutable. *path* doit contenir un chemin absolue ou relatif approprié.

Pour les fonctions `spawnle()`, `spawnlpe()`, `spawnve()`, et `spawnvpe()` (notez qu'elles finissent toutes par « e »), le paramètre *env* doit être un *mapping* qui est utilisé pour définir les variables d'environnement du nouveau processus (celles-ci sont utilisées à la place de l'environnement du nouveau processus). Les fonctions `spawnl()`, `spawnlp()`, `spawnv()`, et `spawnvp()` causent toutes un héritage de l'environnement du processus actuel par le processus fils. Notez que les clefs et les valeurs du dictionnaire *env* doivent être des chaînes de caractères. Des valeurs invalides pour les clefs ou les valeurs met la fonction en échec et renvoie 127.

Par exemple, les appels suivants à `spawnlp()` et `spawnvpe()` sont équivalents :

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```


Disponibilité : Unix, Windows. `spawnlp()`, `spawnlpe()`, `spawnvp()`, et `spawnvpe()` ne sont pas disponibles sur Windows. `spawnle()` et `spawnve()` ne sont pas sécurisés pour les appels concurrents (*thread-safe*) sur Windows, il est conseillé d'utiliser le module `subprocess` à la place.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.P_NOWAIT

os.P_NOWAITO

Valeurs possibles pour le paramètre *mode* de la famille de fonctions `spawn*`. Si l'une de ces valeurs est donnée, les fonctions `spawn*` sortiront dès que le nouveau processus est créé, avec le PID du processus comme valeur de retour.

Disponibilité : Unix, Windows.

os.P_WAIT

Valeur possible pour le paramètre *mode* de la famille de fonctions `spawn*`. Si *mode* est défini par cette valeur, les fonctions `spawn*` ne se terminent pas tant que le nouveau processus n'a pas été complété et renvoient le code de sortie du processus si l'exécution est effectuée avec succès, ou `-signal` si un signal tue le processus.

Disponibilité : Unix, Windows.

os.P_DETACH

os.P_OVERLAY

Valeurs possibles pour le paramètre *mode* de la famille de fonctions `spawn*`. Ces valeurs sont moins portables que celles listées plus haut. `P_DETACH` est similaire à `P_NOWAIT`, mais le nouveau processus est détaché de la console du processus appelant. Si `P_OVERLAY` est utilisé, le processus actuel sera remplacé. La fonction `spawn*` ne sort jamais.

Disponibilité : Windows.

os.startfile(path[, operation])

Lance un fichier avec son application associée.

Quand *operation* n'est pas spécifié ou vaut `'open'`, l'effet est le même qu'un double clic sur le fichier dans Windows Explorer, ou que de passer le nom du fichier en argument du programme `start` depuis l'invite de commande interactif : le fichier est ouvert avec l'application associée à l'extension (s'il y en a une).

Quand une autre *operation* est donnée, ce doit être une « commande-verbe » qui spécifie ce qui devrait être fait avec le fichier. Les verbes habituels documentés par Microsoft sont `'print'` et `'edit'` (qui doivent être utilisés sur des fichiers) ainsi que `'explore'` et `'find'` (qui doivent être utilisés sur des répertoires).

`startfile()` termine dès que l'application associée est lancée. Il n'y a aucune option permettant d'attendre que l'application ne se ferme et aucun moyen de récupérer le statu de sortie de l'application. Le paramètre *path* est relatif au répertoire actuel. Si vous voulez utiliser un chemin absolu, assurez-vous que le premier caractère ne soit pas un slash (`'/'`). La fonction `Win32 ShellExecute()` sous-jacente ne fonctionne pas sinon. Utilisez la fonction `os.path.normpath()` pour vous assurer que le chemin est encodé correctement pour Win32.

Pour réduire le temps système de démarrage de l'interpréteur, la fonction `Win32 ShellExecute()` ne se finit pas tant que cette fonction n'a pas été appelée. Si la fonction ne peut être interprétée, une `NotImplementedError` est levée.

Disponibilité : Windows.

os.system(command)

Exécute la commande (une chaîne de caractères) dans un sous-invite de commandes. C'est implémenté en appelant la fonction standard C `system()` et a les mêmes limitations. Les changements sur `sys.stdin`, etc. ne sont pas reflétés dans l'environnement de la commande exécutée. Si *command* génère une sortie, elle sera envoyée à l'interpréteur standard de flux.

Sur Unix, la valeur de retour est le statut de sortie du processus encodé dans le format spécifié pour `wait()`. Notez que POSIX ne spécifie pas le sens de la valeur de retour de la fonction C `system()`, donc la valeur de retour de la fonction Python est dépendante du système.

Sur Windows, la valeur de retour est celle renvoyée par l'invite de commande système après avoir lancé *command*. L'invite de commande est donné par la variable d'environnement Windows `COMSPEC`. Elle vaut habituellement

`cmd.exe`, qui renvoie l'état de sortie de la commande lancée. Sur les systèmes qui utilisent un invite de commande non-natif, consultez la documentation propre à l'invite.

Le module `subprocess` fournit des outils plus puissants pour générer de nouveaux processus et pour récupérer leur résultat. Il est préférable d'utiliser ce module plutôt que d'utiliser cette fonction. Voir la section *Remplacer les fonctions plus anciennes par le module subprocess* de la documentation du module `subprocess` pour des informations plus précises et utiles.

Disponibilité : Unix, Windows.

os.times()

Renvoie les temps globaux actuels d'exécution du processus. La valeur de retour est un objet avec cinq attributs :

- `user` — le temps utilisateur
- `system` — le temps système
- `children_user` — temps utilisateur de tous les processus fils
- `children_system` — le temps système de tous les processus fils
- `elapsed` — temps écoulé réel depuis un point fixé dans le passé

Pour des raisons de rétro-compatibilité, cet objet se comporte également comme un *tuple* contenant `user`, `system`, `children_user`, `children_system`, et `elapsed` dans cet ordre.

Voir la page de manuel Unix *times(2)* ou la documentation de l'API Windows correspondante. Sur Windows, seuls `user` et `system` sont connus. Les autres attributs sont nuls.

Disponibilité : Unix, Windows.

Modifié dans la version 3.3 : Type de retour changé d'un *tuple* en un objet compatible avec le type *tuple*, avec des attributs nommés.

os.wait()

Attend qu'un processus fils soit complété, et renvoie un *tuple* contenant son PID et son état de sortie : un nombre de 16 bits dont le *byte* de poids faible est le nombre correspondant au signal qui a tué le processus, et dont le *byte* de poids fort est le statut de sortie (si la signal vaut 0). Le bit de poids fort du *byte* de poids faible est mis à 1 si un (fichier système) *core file* a été produit.

Disponibilité : Unix.

os.waitid(idtype, id, options)

Attend qu'un ou plusieurs processus fils soient complétés. *idtypes* peut être `P_PID`, `P_PGID`, ou `P_ALL`. *id* spécifie le PID à attendre. *options* est construit en combinant (avec le OR bit-à-bit) une ou plusieurs des valeurs suivantes : `WEXITED`, `WSTOPPED`, ou `WCONTINUED` et peut additionnellement être combiné avec `WNOHANG` ou `WNOWAIT`. La valeur de retour est un objet représentant les données contenue dans la structure `siginfo_t`, nommées `si_pid`, `si_uid`, `si_signo`, `si_status`, `si_code` ou `None` si `WNOHANG` est spécifié et qu'il n'y a pas d'enfant dans un état que l'on peut attendre.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.P_PID

os.P_PGID

os.P_ALL

Les valeurs possibles pour *idtypes* pour la fonction `waitid()`. Elles affectent l'interprétation de *id*.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.WEXITED

os.WSTOPPED

os.WNOWAIT

Marqueurs qui peuvent être utilisés pour la fonction `waitid()` qui spécifient quel signal attendre du fils.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.CLD_EXITED

os.CLD_DUMPED**os.CLD_TRAPPED****os.CLD_CONTINUED**

Les valeurs possibles pour `si_code` dans le résultat renvoyé par `waitid()`.

Disponibilité : Unix.

Nouveau dans la version 3.3.

os.waitpid(*pid*, *options*)

Les détails de cette fonction diffèrent sur Unix et Windows.

Sur Unix : attend qu'un processus fils donné par son id de processus *pid* soit terminé, et renvoie un *tuple* contenant son PID et son statut de sortie (encodé comme pour `wait()`). La sémantique de cet appel est affecté par la valeur de l'argument entier *options*, qui devrait valoir 0 pour les opérations normales.

Si *pid* est plus grand que 0, `waitpid()` introduit une requête pour des informations sur le statut du processus spécifié. Si *pid* vaut 0, la requête est pour le statut de tous les fils dans le groupe de processus du processus actuel. Si *pid* vaut -1, la requête concerne tous les processus fils du processus actuel. Si *pid* est inférieur à -1, une requête est faite pour le statut de chaque processus du groupe de processus donné par `-pid` (la valeur absolue de *pid*).

Une `OSError` est levée avec la valeur de *errno* quand l'appel système renvoie -1.

Sur Windows : attend que qu'un processus donné par l'identificateur de processus (*process handle*) *pid* soit complété, et renvoie un *tuple* contenant *pid* et son statut de sortie décalé de 8 bits vers la gauche (le décalage rend l'utilisation multiplate-forme plus facile). Un *pid* inférieur ou égal à 0 n'a aucune signification particulière sur Windows, et lève une exception. L'argument entier *options* n'a aucun effet. *pid* peut faire référence à tout processus dont l'identifiant est connu pas nécessairement un processus fils. Les fonctions *spawn** appelées avec `P_NOWAIT` renvoient des identificateurs de processus appropriés.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

os.wait3(*options*)

Similaire à la fonction `waitpid()`, si ce n'est que qu'aucun argument *id* n'est donné, et qu'un triplet contenant l'identifiant du processus fils, son statut de sortie, et une information sur l'utilisation des ressources est renvoyé. Référez-vous à `resource.getrusage()` pour des détails sur les informations d'utilisation des ressources. L'argument *options* est le même que celui fourni à `waitpid()` et `wait4()`.

Disponibilité : Unix.

os.wait4(*pid*, *options*)

Similaire à `waitpid()`, si ce n'est qu'un triplet contenant l'identifiant du processus fils, son statut de sortie, et des informations d'utilisation des ressources est renvoyé. Référez-vous à `resource.getrusage()` pour des détails sur les informations d'utilisation des ressources. Les arguments de `wait4()` sont les mêmes que ceux fournis à `waitpid()`.

Disponibilité : Unix.

os.WNOHANG

L'option de `waitpid()` pour terminer immédiatement si aucun statut de processus fils n'est disponible dans l'immédiat. La fonction renvoie (0, 0) dans ce cas.

Disponibilité : Unix.

os.WCONTINUED

Cette option cause les processus fils à être reportés s'ils ont été continués après un arrêt du *job control* depuis leurs derniers reports de statuts.

Disponibilité : certains systèmes Unix.

os.WUNTRACED

Cette option cause les processus fils à être reportés s'ils ont été stoppés mais que leur état actuel n'a pas été reporté depuis qu'ils ont été stoppés.

Disponibilité : Unix.

Les fonctions suivantes prennent un code de statu tel que renvoyé par `system()`, `wait()`, ou `waitpid()` en paramètre. Ils peuvent être utilisés pour déterminer la disposition d'un processus.

- **`os.WCOREDUMP`** (*status*)
Renvoie `True` si un vidage système (*core dump*) a été généré pour le processus, sinon, renvoie `False`.
Disponibilité : Unix.
- **`os.WIFCONTINUED`** (*status*)
Renvoie `True` si le processus a été continué après un arrêt du *job control*, renvoie `False` autrement.
Disponibilité : Unix.
- **`os.WIFSTOPPED`** (*status*)
Renvoie `True` si le processus a été arrêté, sinon renvoie `False`.
Disponibilité : Unix.
- **`os.WIFSIGNALED`** (*status*)
Renvoie `True` si le processus s'est terminé à cause d'un signal, sinon, renvoie `False`.
Disponibilité : Unix.
- **`os.WIFEXITED`** (*status*)
Renvoie `True` si le processus s'est terminé en faisant un appel système `exit(2)`, sinon, renvoie `False`.
Disponibilité : Unix.
- **`os.WEXITSTATUS`** (*status*)
Si `WIFEXITED(status)` vaut `True`, renvoie le paramètre entier de l'appel système `exit(2)`. Sinon, la valeur de retour n'a pas de signification.
Disponibilité : Unix.
- **`os.WSTOPSIG`** (*status*)
Renvoie le signal qui a causé l'arrêt du processus.
Disponibilité : Unix.
- **`os.WTERMSIG`** (*status*)
Renvoie le signal qui a amené le processus à quitter.
Disponibilité : Unix.

16.1.7 Interface pour l'ordonnanceur

Ces fonctions contrôlent un processus se voit allouer du temps de processus par le système d'exploitation. Elles ne sont disponibles que sur certaines plate-formes Unix. Pour des informations plus détaillées, consultez les pages de manuels Unix.

Nouveau dans la version 3.3.

Les polices d'ordonnancement suivantes sont exposées si elles sont gérées par le système d'exploitation.

- **`os.SCHED_OTHER`**
La police d'ordonnancement par défaut.
- **`os.SCHED_BATCH`**
Police d'ordonnancement pour les processus intensifs en utilisation du processeur. Cette police essaye de préserver l'interactivité pour le reste de l'ordinateur.
- **`os.SCHED_IDLE`**
Police d'ordonnancement pour les tâches de fond avec une priorité extrêmement faible.
- **`os.SCHED_SPORADIC`**
Police d'ordonnancement pour des programmes serveurs sporadiques.

os.SCHED_FIFO

Une police d'ordonnancement *FIFO* (dernier arrivé, premier servi).

os.SCHED_RR

Une police d'ordonnancement *round-robin* (tourniquet).

os.SCHED_RESET_ON_FORK

Cette option peut combiner différentes politiques d'ordonnancement avec un OU bit-à-bit. Quand un processus avec cette option se dédouble, la politique d'ordonnancement et la priorité du processus fils sont remises aux valeurs par défaut.

class os.sched_param (sched_priority)

Cette classe représente des paramètres d'ordonnancement réglables utilisés pour `sched_setparam()`, `sched_setscheduler()`, et `sched_getparam()`. Un objet de ce type est immuable.

Pour le moment, il n'y a qu'un seul paramètre possible :

sched_priority

La priorité d'ordonnancement pour une police d'ordonnancement.

os.sched_get_priority_min (policy)

Récupère la valeur minimum pour une priorité pour la police *policy*. *policy* est une des constantes de police définies ci-dessus.

os.sched_get_priority_max (policy)

Récupère la valeur maximum pour une priorité pour la police *policy*. *policy* est une des constantes de police définies ci-dessus.

os.sched_setscheduler (pid, policy, param)

Définit la police d'ordonnancement pour le processus de PID *pid*. Un *pid* de 0 signifie le processus appelant. *policy* est une des constantes de police définies ci-dessus. *param* est une instance de la classe `sched_param`.

os.sched_getscheduler (pid)

Renvoie la police d'ordonnancement pour le processus de PID *pid*. Un *pid* de 0 signifie le processus appelant. Le résultat est une des constantes de police définies ci-dessus.

os.sched_setparam (pid, param)

Définit un paramètre d'ordonnancement pour le processus de PID *pid*. Un *pid* de 0 signifie le processus appelant. *param* est une instance de `sched_param`.

os.sched_getparam (pid)

Renvoie les paramètres d'ordonnancement dans une de `sched_param` pour le processus de PID *pid*. Un *pid* de 0 signifie le processus appelant.

os.sched_rr_get_interval (pid)

Renvoie le quantum de temps du *round-robin* (en secondes) pour le processus de PID *pid*. Un *pid* de 0 signifie le processus appelant.

os.sched_yield ()

Abandonne volontairement le processeur.

os.sched_setaffinity (pid, mask)

Restreint le processus de PID *pid* (ou le processus actuel si *pid* vaut 0) à un ensemble de CPUs. *mask* est un itérable d'entiers représentant l'ensemble de CPUs auquel le processus doit être restreint.

os.sched_getaffinity (pid)

Renvoie l'ensemble de CPUs auquel le processus de PID *pid* (ou le processus actuel si *pid* vaut 0) est restreint.

16.1.8 Diverses informations sur le système

`os.confstr(name)`

Renvoie les valeurs de configuration en chaînes de caractères. *name* spécifie la valeur de configuration à récupérer. Ce peut être une chaîne de caractères représentant le nom d'une valeur système définie dans un nombre de standards (POSIX, Unix 95, Unix 98, et d'autres). Certaines plate-formes définissent des noms supplémentaires également. Les noms connus par le système d'exploitation hôte sont données dans les clefs du dictionnaire `confstr_names`. Pour les variables de configuration qui ne sont pas incluses dans ce *mapping*, passer un entier pour *name* est également accepté.

Si la valeur de configuration spécifiée par *name* n'est pas définie, `None` est renvoyé.

Si *name* est une chaîne de caractères et n'est pas connue, une `ValueError` est levée. Si une valeur spécifique pour *name* n'est pas gérée par le système hôte, même si elle est incluse dans `confstr_names`, une `OSError` est levée avec `errno.EINVAL` pour numéro d'erreur.

Disponibilité : Unix.

`os.confstr_names`

Dictionnaire liant les noms acceptés par `confstr()` aux valeurs entières définies pour ces noms par le système d'exploitation hôte. Cela peut être utilisé pour déterminer l'ensemble des noms connus du système.

Disponibilité : Unix.

`os.cpu_count()`

Renvoie le nombre de CPUs dans le système. Renvoie `None` si indéterminé.

Ce nombre n'est pas équivalent au nombre de CPUs que le processus courant peut utiliser. Le nombre de CPUs utilisables peut être obtenu avec `len(os.sched_getaffinity(0))`

Nouveau dans la version 3.4.

`os.getloadavg()`

Renvoie le nombre de processus dans la file d'exécution du système en moyenne dans les dernières 1, 5, et 15 minutes, ou lève une `OSError` si la charge moyenne est impossible à récupérer.

Disponibilité : Unix.

`os.sysconf(name)`

Renvoie les valeurs de configuration en nombres entiers. Si la valeur de configuration définie par *name* n'est pas spécifiée, `-1` est renvoyé. Les commentaires concernant le paramètre *name* de `confstr()` s'appliquent également ici, le dictionnaire qui fournit les informations sur les noms connus est donné par `sysconf_names`.

Disponibilité : Unix.

`os.sysconf_names`

Dictionnaire liant les noms acceptés par `sysconf()` aux valeurs entières définies pour ces noms par le système d'exploitation hôte. Cela peut être utilisé pour déterminer l'ensemble des noms connus du système.

Disponibilité : Unix.

Les valeurs suivantes sont utilisées pour gérer les opérations de manipulations de chemins. Elles sont définies pour toutes les plate-formes.

Des opérations de plus haut niveau sur les chemins sont définies dans le module `os.path`.

`os.curdir`

La chaîne de caractère constante utilisée par le système d'exploitation pour référencer le répertoire actuel. Ça vaut `'.'` pour Windows et POSIX. Également disponible par `os.path`.

`os.pardir`

La chaîne de caractère constante utilisée par le système d'exploitation pour référencer le répertoire parent. Ça vaut `'..'` pour Windows et POSIX. Également disponible par `os.path`.

`os.sep`

Le caractère utilisé par le système d'exploitation pour séparer les composantes des chemins. C'est `'/'` pour POSIX, et `'\\'` pour Windows. Notez que ce n'est pas suffisant pour pouvoir analyser ou concaténer correctement

des chemins (utilisez alors `os.path.split()` et `os.path.join()`), mais ça peut s'avérer utile occasionnellement. Également disponible par `os.path`.

`os.altsep`

Un caractère alternatif utilisé par le système d'exploitation pour séparer les composantes des chemins, ou `None` si un seul séparateur existe. Ça vaut `'/'` sur Windows où `sep` est un antislash `'\'`. Également disponible par `os.path`.

`os.extsep`

Le caractère qui sépare la base du nom d'un fichier et son extension. Par exemple, le `'.'` de `os.py`. Également disponible par `os.path`.

`os.pathsep`

Le caractère conventionnellement utilisé par le système d'exploitation pour séparer la recherche des composantes de chemin (comme dans la variable d'environnement `PATH`). Cela vaut `':'` pour POSIX, ou `';'` pour Windows. Également disponible par `os.path`.

`os.defpath`

Le chemin de recherche par défaut utilisé par `exec*` et `spawn*` si l'environnement n'a pas une clef `'PATH'`. Également disponible par `os.path`.

`os.linesep`

La chaîne de caractères utilisée pour séparer (ou plutôt pour terminer) les lignes sur la plate-forme actuelle. Ce peut être un caractère unique (comme `'\n'` pour POSIX,) ou plusieurs caractères (comme `'\r\n'` pour Windows). N'utilisez pas `os.linesep` comme terminateur de ligne quand vous écrivez dans un fichier ouvert en mode *texte* (par défaut). Utilisez un unique `'\n'` à la place, sur toutes les plate-formes.

`os.devnull`

Le chemin de fichier du périphérique *null*. Par exemple : `'/dev/null '` pour POSIX, `'nul '` pour Windows. Également disponible par `os.path`.

`os.RTLD_LAZY`

`os.RTLD_NOW`

`os.RTLD_GLOBAL`

`os.RTLD_LOCAL`

`os.RTLD_NODELETE`

`os.RTLD_NOLOAD`

`os.RTLD_DEEPBIND`

Marqueurs à utiliser avec les fonctions `setdlopenflags()` et `getdlopenflags()`. Voir les pages de manuel Unix `dlopen(3)` pour les différences de significations entre les marqueurs.

Nouveau dans la version 3.3.

16.1.9 Nombres aléatoires

`os.getrandom(size, flags=0)`

Obtient *size* octets aléatoires. La fonction renvoie éventuellement moins d'octets que demandé.

Ces octets peuvent être utilisés pour initialiser un générateur de nombres aléatoires dans l'espace utilisateur ou pour des raisons cryptographiques.

`getrandom()` se base sur l'entropie rassemblée depuis les pilotes des périphériques et autres sources de bruits de l'environnement. La lecture de grosses quantités de données aura un impact négatif sur les autres utilisateurs des périphériques `/dev/random` et `/dev/urandom`.

L'argument *flags* est un champ de bits qui peut contenir zéro ou plus des valeurs suivantes combinées avec un OU bit-à-bit : `os.GRND_RANDOM` et `GRND_NONBLOCK`.

Voir aussi la [page de manuel Linux pour getrandom\(\)](#).

Disponibilité : Linux 3.17 et ultérieures.

Nouveau dans la version 3.6.

os.urandom(*size*)

Renvoie une chaîne de *size* octets aléatoires utilisable dans un cadre cryptographique.

Cette fonction renvoie des octets aléatoires depuis une source spécifique à l'OS. Les données renvoyées sont censées être suffisamment imprévisibles pour les applications cryptographiques, bien que la qualité dépende de l'implémentation du système.

Sous Linux, si l'appel système `getrandom()` est disponible, il est utilisé en mode bloquant : il bloque jusqu'à ce que la réserve d'entropie d'*urandom* soit initialisée (128 bits d'entropie sont collectés par le noyau). Voir la [PEP 524](#) pour plus d'explications. Sous Linux, la fonction `getrandom()` peut être utilisée pour obtenir des octets aléatoires en mode non-bloquant (avec l'option `GRND_NONBLOCK`) ou attendre jusqu'à ce que la réserve d'entropie d'*urandom* soit initialisée.

Sur un système de type UNIX, les octets aléatoires sont lus depuis le périphérique `/dev/urandom`. Si le périphérique `/dev/urandom` n'est pas disponible ou n'est pas lisible, l'exception `NotImplementedError` est levée.

Sous Windows, `CryptGenRandom()` est utilisée.

Voir aussi :

Le module `secrets` fournit des fonctions de plus haut niveau. Pour une interface facile à utiliser du générateur de nombres aléatoires fourni par votre plate-forme, veuillez regarder `random.SystemRandom`.

Modifié dans la version 3.6.0 : Sous Linux, `getrandom()` est maintenant utilisé en mode bloquant pour renforcer la sécurité.

Modifié dans la version 3.5.2 : Sous Linux, si l'appel système `getrandom()` bloque (la réserve d'entropie d'*urandom* n'est pas encore initialisée), réalise à la place une lecture de `/dev/urandom`.

Modifié dans la version 3.5 : Sur Linux 3.17 et plus récent, l'appel système `getrandom()` est maintenant utilisé quand il est disponible. Sur OpenBSD 5.6 et plus récent, la fonction C `getentropy()` est utilisée. Ces fonctions évitent l'utilisation interne d'un descripteur de fichier.

os.GRND_NONBLOCK

Par défaut, quand elle lit depuis `/dev/random`, `getrandom()` bloque si aucun octet aléatoire n'est disponible, et quand elle lit depuis `/dev/urandom`, elle bloque si la réserve d'entropie n'a pas encore été initialisée.

Si l'option `GRND_NONBLOCK` est activée, `getrandom()` ne bloque pas dans ces cas, mais lève immédiatement une `BlockingIOError`.

Nouveau dans la version 3.6.

os.GRND_RANDOM

Si ce bit est activé, les octets aléatoires sont puisés depuis `/dev/random` plutôt que `/dev/urandom`.

Nouveau dans la version 3.6.

16.2 io — Core tools for working with streams

Source code : [Lib/io.py](#)

16.2.1 Aperçu

The `io` module provides Python's main facilities for dealing with various types of I/O. There are three main types of I/O : *text I/O*, *binary I/O* and *raw I/O*. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a *file object*. Other common terms are *stream* and *file-like object*.

Independent of its category, each concrete stream object will also have various capabilities : it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example giving a *str* object to the `write()` method of a binary stream will raise a `TypeError`. So will giving a *bytes* object to the `write()` method of a text stream.

Modifié dans la version 3.3 : Operations that used to raise *IOError* now raise *OSError*, since *IOError* is now an alias of *OSError*.

Text I/O

Text I/O expects and produces *str* objects. This means that whenever the backing store is natively made of bytes (such as in the case of a file), encoding and decoding of data is made transparently as well as optional translation of platform-specific newline characters.

The easiest way to create a text stream is with `open()`, optionally specifying an encoding :

```
f = open("myfile.txt", "r", encoding="utf-8")
```

In-memory text streams are also available as *StringIO* objects :

```
f = io.StringIO("some initial text data")
```

The text stream API is described in detail in the documentation of *TextIOBase*.

Binary I/O

Binary I/O (also called *buffered I/O*) expects *bytes-like objects* and produces *bytes* objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.

The easiest way to create a binary stream is with `open()` with 'b' in the mode string :

```
f = open("myfile.jpg", "rb")
```

In-memory binary streams are also available as *BytesIO* objects :

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

The binary stream API is described in detail in the docs of *BufferedIOBase*.

Other library modules may provide additional ways to create text or binary streams. See `socket.socket.makefile()` for example.

Raw I/O

Raw I/O (also called *unbuffered I/O*) is generally used as a low-level building-block for binary and text streams ; it is rarely useful to directly manipulate a raw stream from user code. Nevertheless, you can create a raw stream by opening a file in binary mode with buffering disabled :

```
f = open("myfile.jpg", "rb", buffering=0)
```

The raw stream API is described in detail in the docs of *RawIOBase*.

16.2.2 High-level Module Interface

`io.DEFAULT_BUFFER_SIZE`

An int containing the default buffer size used by the module's buffered I/O classes. `open()` uses the file's `blksize` (as obtained by `os.stat()`) if possible.

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

This is an alias for the builtin `open()` function.

exception `io.BlockingIOError`

This is a compatibility alias for the builtin `BlockingIOError` exception.

exception `io.UnsupportedOperation`

An exception inheriting `OSError` and `ValueError` that is raised when an unsupported operation is called on a stream.

In-memory streams

It is also possible to use a *str* or *bytes-like object* as a file for both reading and writing. For strings `StringIO` can be used like a file opened in text mode. `BytesIO` can be used like a file opened in binary mode. Both provide full read-write capabilities with random access.

Voir aussi :

`sys` contains the standard IO streams : `sys.stdin`, `sys.stdout`, and `sys.stderr`.

16.2.3 Class hierarchy

The implementation of I/O streams is organized as a hierarchy of classes. First *abstract base classes* (ABCs), which are used to specify the various categories of streams, then concrete classes providing the standard stream implementations.

Note : The abstract base classes also provide default implementations of some methods in order to help implementation of concrete stream classes. For example, `BufferedIOBase` provides unoptimized implementations of `readinto()` and `readline()`.

At the top of the I/O hierarchy is the abstract base class `IOBase`. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise `UnsupportedOperation` if they do not support a given operation.

The `RawIOBase` ABC extends `IOBase`. It deals with the reading and writing of bytes to a stream. `FileIO` subclasses `RawIOBase` to provide an interface to files in the machine's file system.

The `BufferedIOBase` ABC deals with buffering on a raw byte stream (`RawIOBase`). Its subclasses, `BufferedWriter`, `BufferedReader`, and `BufferedRWPair` buffer streams that are readable, writable, and both readable and writable. `BufferedRandom` provides a buffered interface to random access streams. Another `BufferedIOBase` subclass, `BytesIO`, is a stream of in-memory bytes.

The `TextIOBase` ABC, another subclass of `IOBase`, deals with streams whose bytes represent text, and handles encoding and decoding to and from strings. `TextIOWrapper`, which extends it, is a buffered text interface to a buffered raw stream (`BufferedIOBase`). Finally, `StringIO` is an in-memory stream for text.

Argument names are not part of the specification, and only the arguments of `open()` are intended to be used as keyword arguments.

The following table summarizes the ABCs provided by the `io` module :

ABC	Inherits	Stub Methods	Mixin Methods and Properties
<i>IOBase</i>		<code>fileno</code> , <code>seek</code> , et <code>truncate</code>	<code>close</code> , <code>closed</code> , <code>__enter__</code> , <code>__exit__</code> , <code>flush</code> , <code>isatty</code> , <code>__iter__</code> , <code>__next__</code> , <code>readable</code> , <code>readline</code> , <code>readlines</code> , <code>seekable</code> , <code>tell</code> , <code>writable</code> , and <code>writelines</code>
<i>RawIOBase</i>	<i>IOBase</i>	<code>readinto</code> et <code>write</code>	Inherited <i>IOBase</i> methods, <code>read</code> , and <code>readall</code>
<i>BufferedIOBase</i>	<i>IOBase</i>	<code>detach</code> , <code>read</code> , <code>read1</code> , et <code>write</code>	Inherited <i>IOBase</i> methods, <code>readinto</code> , and <code>readinto1</code>
<i>TextIOBase</i>	<i>IOBase</i>	<code>detach</code> , <code>read</code> , <code>readline</code> , et <code>write</code>	Inherited <i>IOBase</i> methods, <code>encoding</code> , <code>errors</code> , and <code>newlines</code>

I/O Base Classes

`class io.IOBase`

The abstract base class for all I/O classes, acting on streams of bytes. There is no public constructor.

This class provides empty abstract implementations for many methods that derived classes can override selectively; the default implementations represent a file that cannot be read, written or seeked.

Even though *IOBase* does not declare `read()`, `readinto()`, or `write()` because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a *ValueError* (or *UnsupportedOperation*) when operations they do not support are called.

The basic type used for binary data read from or written to a file is *bytes*. Other *bytes-like objects* are accepted as method arguments too. In some cases, such as `readinto()`, a writable object such as *bytearray* is required. Text I/O classes work with *str* data.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise *ValueError* in this case.

IOBase (and its subclasses) supports the iterator protocol, meaning that an *IOBase* object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding character strings). See `readline()` below.

IOBase is also a context manager and therefore supports the `with` statement. In this example, *file* is closed after the `with` statement's suite is finished—even if an exception occurs :

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase provides these data attributes and methods :

`close()`

Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a *ValueError*.

As a convenience, it is allowed to call this method more than once ; only the first call, however, will have an effect.

`closed`

True if the stream is closed.

`fileno()`

Return the underlying file descriptor (an integer) of the stream if it exists. An *OSError* is raised if the IO object does not use a file descriptor.

flush()

Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

isatty()

Return True if the stream is interactive (i.e., connected to a terminal/tty device).

readable()

Return True if the stream can be read from. If False, `read()` will raise `IOError`.

readline (size=-1)

Read and return one line from the stream. If *size* is specified, at most *size* bytes will be read.

The line terminator is always `b'\n'` for binary files; for text files, the *newline* argument to `open()` can be used to select the line terminator(s) recognized.

readlines (hint=-1)

Read and return a list of lines from the stream. *hint* can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds *hint*.

Note that it's already possible to iterate on file objects using `for line in file: ...` without calling `file.readlines()`.

seek (offset[, whence])

Change the stream position to the given byte *offset*. *offset* is interpreted relative to the position indicated by *whence*. The default value for *whence* is `SEEK_SET`. Values for *whence* are:

- `SEEK_SET` or 0 – start of the stream (the default); *offset* should be zero or positive
- `SEEK_CUR` or 1 – current stream position; *offset* may be negative
- `SEEK_END` or 2 – end of the stream; *offset* is usually negative

Return the new absolute position.

Nouveau dans la version 3.1 : The `SEEK_*` constants.

Nouveau dans la version 3.3 : Some operating systems could support additional values, like `os.SEEK_HOLE` or `os.SEEK_DATA`. The valid values for a file could depend on it being open in text or binary mode.

seekable()

Return True if the stream supports random access. If False, `seek()`, `tell()` and `truncate()` will raise `IOError`.

tell()

Return the current stream position.

truncate (size=None)

Resize the stream to the given *size* in bytes (or the current position if *size* is not specified). The current stream position isn't changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled). The new file size is returned.

Modifié dans la version 3.5 : Windows will now zero-fill files when extending.

writable()

Return True if the stream supports writing. If False, `write()` and `truncate()` will raise `IOError`.

writelines (lines)

Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

__del__()

Prepare for object destruction. `IOBase` provides a default implementation of this method that calls the instance's `close()` method.

class io.RawIOBase

Base class for raw binary I/O. It inherits `IOBase`. There is no public constructor.

Raw binary I/O typically provides low-level access to an underlying OS device or API, and does not try to encapsulate it in high-level primitives (this is left to Buffered I/O and Text I/O, described later in this page).

In addition to the attributes and methods from `IOBase`, `RawIOBase` provides the following methods:

read (size=-1)

Read up to *size* bytes from the object and return them. As a convenience, if *size* is unspecified or -1, all bytes

until EOF are returned. Otherwise, only one system call is ever made. Fewer than *size* bytes may be returned if the operating system call returns fewer than *size* bytes.

If 0 bytes are returned, and *size* was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, *None* is returned.

The default implementation defers to *readall()* and *readinto()*.

readall()

Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

readinto(*b*)

Read bytes into a pre-allocated, writable *bytes-like object* *b*, and return the number of bytes read. If the object is in non-blocking mode and no bytes are available, *None* is returned.

write(*b*)

Write the given *bytes-like object*, *b*, to the underlying raw stream, and return the number of bytes written. This can be less than the length of *b* in bytes, depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode. *None* is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

class io.BufferedReader

Base class for binary streams that support some kind of buffering. It inherits *IOBase*. There is no public constructor.

The main difference with *RawIOBase* is that methods *read()*, *readinto()* and *write()* will try (respectively) to read as much input as requested or to consume all given output, at the expense of making perhaps more than one system call.

In addition, those methods can raise *BlockingIOError* if the underlying raw stream is in non-blocking mode and cannot take or give enough data; unlike their *RawIOBase* counterparts, they will never return *None*.

Besides, the *read()* method does not have a default implementation that defers to *readinto()*.

A typical *BufferedReader* implementation should not inherit from a *RawIOBase* implementation, but wrap one, like *BufferedWriter* and *BufferedReader* do.

BufferedReader provides or overrides these methods and attribute in addition to those from *IOBase*:

raw

The underlying raw stream (a *RawIOBase* instance) that *BufferedReader* deals with. This is not part of the *BufferedReader* API and may not exist on some implementations.

detach()

Separate the underlying raw stream from the buffer and return it.

After the raw stream has been detached, the buffer is in an unusable state.

Some buffers, like *BytesIO*, do not have the concept of a single raw stream to return from this method. They raise *UnsupportedOperation*.

Nouveau dans la version 3.1.

read(*size=-1*)

Read and return up to *size* bytes. If the argument is omitted, *None*, or negative, data is read and returned until EOF is reached. An empty *bytes* object is returned if the stream is already at EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.

A *BlockingIOError* is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

read1(*size=-1*)

Read and return up to *size* bytes, with at most one call to the underlying raw stream's *read()* (or *readinto()*) method. This can be useful if you are implementing your own buffering on top of a *BufferedReader* object.

readinto(*b*)

Read bytes into a pre-allocated, writable *bytes-like object* *b* and return the number of bytes read.

Like `read()`, multiple reads may be issued to the underlying raw stream, unless the latter is interactive.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

readinto1 (*b*)

Read bytes into a pre-allocated, writable *bytes-like object* *b*, using at most one call to the underlying raw stream's `read()` (or `readinto()`) method. Return the number of bytes read.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

Nouveau dans la version 3.5.

write (*b*)

Write the given *bytes-like object*, *b*, and return the number of bytes written (always equal to the length of *b* in bytes, since if the write fails an `OSError` will be raised). Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

When in non-blocking mode, a `BlockingIOError` is raised if the data needed to be written to the raw stream but it couldn't accept all the data without blocking.

The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

Raw File I/O

class `io.FileIO` (*name*, *mode*='r', *closefd*=True, *opener*=None)

FileIO represents an OS-level file containing bytes data. It implements the *RawIOBase* interface (and therefore the *IOBase* interface, too).

The *name* can be one of two things :

- a character string or *bytes* object representing the path to the file which will be opened. In this case *closefd* must be True (the default) otherwise an error will be raised.
- an integer representing the number of an existing OS-level file descriptor to which the resulting *FileIO* object will give access. When the *FileIO* object is closed this *fd* will be closed as well, unless *closefd* is set to False.

The *mode* can be 'r', 'w', 'x' or 'a' for reading (default), writing, exclusive creation or appending. The file will be created if it doesn't exist when opened for writing or appending ; it will be truncated when opened for writing. `FileExistsError` will be raised if it already exists when opened for creating. Opening a file for creating implies writing, so this mode behaves in a similar way to 'w'. Add a '+' to the mode to allow simultaneous reading and writing.

The `read()` (when called with a positive argument), `readinto()` and `write()` methods on this class will only make one system call.

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

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

See the `open()` built-in function for examples on using the *opener* parameter.

Modifié dans la version 3.3 : The *opener* parameter was added. The 'x' mode was added.

Modifié dans la version 3.4 : Il n'est plus possible d'hériter de *file*.

In addition to the attributes and methods from *IOBase* and *RawIOBase*, *FileIO* provides the following data attributes :

mode

The mode as given in the constructor.

name

The file name. This is the file descriptor of the file when no name is given in the constructor.

Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

class `io.BytesIO([initial_bytes])`

A stream implementation using an in-memory bytes buffer. It inherits *BufferedIOBase*. The buffer is discarded when the *close()* method is called.

The optional argument *initial_bytes* is a *bytes-like object* that contains initial data.

BytesIO provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

getbuffer()

Return a readable and writable view over the contents of the buffer without copying them. Also, mutating the view will transparently update the contents of the buffer:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

Note : As long as the view exists, the *BytesIO* object cannot be resized or closed.

Nouveau dans la version 3.2.

getvalue()

Return *bytes* containing the entire contents of the buffer.

read1()

In *BytesIO*, this is the same as *read()*.

readinto1()

In *BytesIO*, this is the same as *readinto()*.

Nouveau dans la version 3.5.

class `io.BufferedReader(raw, buffer_size=DEFAULT_BUFFER_SIZE)`

A buffer providing higher-level access to a readable, sequential *RawIOBase* object. It inherits *BufferedIOBase*. When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

The constructor creates a *BufferedReader* for the given readable *raw* stream and *buffer_size*. If *buffer_size* is omitted, *DEFAULT_BUFFER_SIZE* is used.

BufferedReader provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

peek([size])

Return bytes from the stream without advancing the position. At most one single read on the raw stream is done to satisfy the call. The number of bytes returned may be less or more than requested.

read([size])

Read and return *size* bytes, or if *size* is not given or negative, until EOF or if the read call would block in non-blocking mode.

read1(size)

Read and return up to *size* bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

class `io.BufferedWriter(raw, buffer_size=DEFAULT_BUFFER_SIZE)`

A buffer providing higher-level access to a writable, sequential *RawIOBase* object. It inherits *BufferedIOBase*. When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying *RawIOBase* object under various conditions, including:

- when the buffer gets too small for all pending data;

- when `flush()` is called;
- when a `seek()` is requested (for `BufferedRandom` objects);
- when the `BufferedWriter` object is closed or destroyed.

The constructor creates a `BufferedWriter` for the given writeable `raw` stream. If the `buffer_size` is not given, it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedWriter` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase`:

flush()

Force bytes held in the buffer into the raw stream. A `BlockingIOError` should be raised if the raw stream blocks.

write(b)

Write the *bytes-like object*, `b`, and return the number of bytes written. When in non-blocking mode, a `BlockingIOError` is raised if the buffer needs to be written out but the raw stream blocks.

class `io.BufferedRandom` (`raw`, `buffer_size=DEFAULT_BUFFER_SIZE`)

A buffered interface to random access streams. It inherits `BufferedReader` and `BufferedWriter`, and further supports `seek()` and `tell()` functionality.

The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the `buffer_size` is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedRandom` is capable of anything `BufferedReader` or `BufferedWriter` can do.

class `io.BufferedRWPair` (`reader`, `writer`, `buffer_size=DEFAULT_BUFFER_SIZE`)

A buffered I/O object combining two unidirectional `RawIOBase` objects – one readable, the other writeable – into a single bidirectional endpoint. It inherits `BufferedIOBase`.

`reader` and `writer` are `RawIOBase` objects that are readable and writeable respectively. If the `buffer_size` is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedRWPair` implements all of `BufferedIOBase`'s methods except for `detach()`, which raises `UnsupportedOperation`.

Avertissement : `BufferedRWPair` does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use `BufferedRandom` instead.

Text I/O

class `io.TextIOBase`

Base class for text streams. This class provides a character and line based interface to stream I/O. There is no `readinto()` method because Python's character strings are immutable. It inherits `IOBase`. There is no public constructor.

`TextIOBase` provides or overrides these data attributes and methods in addition to those from `IOBase`:

encoding

The name of the encoding used to decode the stream's bytes into strings, and to encode strings into bytes.

errors

The error setting of the decoder or encoder.

newlines

A string, a tuple of strings, or `None`, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

buffer

The underlying binary buffer (a `BufferedIOBase` instance) that `TextIOBase` deals with. This is not part of the `TextIOBase` API and may not exist in some implementations.

detach()

Separate the underlying binary buffer from the `TextIOBase` and return it.

After the underlying buffer has been detached, the `TextIOBase` is in an unusable state.

Some `TextIOBase` implementations, like `StringIO`, may not have the concept of an underlying buffer and calling this method will raise `UnsupportedOperation`.

Nouveau dans la version 3.1.

read (*size=-1*)

Read and return at most *size* characters from the stream as a single `str`. If *size* is negative or `None`, reads until EOF.

readline (*size=-1*)

Read until newline or EOF and return a single `str`. If the stream is already at EOF, an empty string is returned.

If *size* is specified, at most *size* characters will be read.

seek (*offset* [, *whence*])

Change the stream position to the given *offset*. Behaviour depends on the *whence* parameter. The default value for *whence* is `SEEK_SET`.

— `SEEK_SET` or 0 : seek from the start of the stream (the default); *offset* must either be a number returned by `TextIOBase.tell()`, or zero. Any other *offset* value produces undefined behaviour.

— `SEEK_CUR` or 1 : « seek » to the current position; *offset* must be zero, which is a no-operation (all other values are unsupported).

— `SEEK_END` or 2 : seek to the end of the stream; *offset* must be zero (all other values are unsupported).

Return the new absolute position as an opaque number.

Nouveau dans la version 3.1 : The `SEEK_*` constants.

tell ()

Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write (*s*)

Write the string *s* to the stream and return the number of characters written.

class `io.TextIOWrapper` (*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)

A buffered text stream over a `BufferedIOBase` binary stream. It inherits `TextIOBase`.

encoding gives the name of the encoding that the stream will be decoded or encoded with. It defaults to `locale.getpreferredencoding(False)`.

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass `'strict'` to raise a `ValueError` exception if there is an encoding error (the default of `None` has the same effect), or pass `'ignore'` to ignore errors. (Note that ignoring encoding errors can lead to data loss.) `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data. `'backslashreplace'` causes malformed data to be replaced by a backslashed escape sequence. When writing, `'xmlcharrefreplace'` (replace with the appropriate XML character reference) or `'namereplace'` (replace with `\N{...}` escape sequences) can be used. Any other error handling name that has been registered with `codecs.register_error()` is also valid.

newline controls how line endings are handled. It can be `None`, `' '`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows :

- Lors de la lecture, si *newline* est `None`, le mode *universal newlines* est activé. Les lignes lues peuvent terminer par `'\n'`, `'\r'`, ou `'\r\n'`, qui sont remplacés par `'\n'`, avant d'être données à l'appelant. S'il vaut `' '`, le mode *universal newline* est activé mais les fin de lignes ne sont pas remplacés. S'il a n'importe quel autre valeur autorisée, les lignes sont seulement terminées par la chaîne donnée, qui est rendue tel qu'elle.
- Lors de l'écriture, si *newline* est `None`, chaque `'\n'` est remplacé par le séparateur de lignes par défaut du système `os.linesep`. Si *newline* est `*` ou `'\n'` aucun remplacement n'est effectué. Si *newline* est un autre caractère valide, chaque `'\n'` sera remplacé par la chaîne donnée.

If *line_buffering* is `True`, `flush()` is implied when a call to write contains a newline character or a carriage return.

If *write_through* is `True`, calls to `write()` are guaranteed not to be buffered : any data written on the `TextIOWrapper` object is immediately handled to its underlying binary *buffer*.

Modifié dans la version 3.3 : The `write_through` argument has been added.

Modifié dans la version 3.3 : The default `encoding` is now `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. Don't change temporarily the locale encoding using `locale.setlocale()`, use the current locale encoding instead of the user preferred encoding.

`TextIOWrapper` provides one attribute in addition to those of `TextIOBase` and its parents :

line_buffering

Whether line buffering is enabled.

class `io.StringIO` (`initial_value=""`, `newline='\n'`)

An in-memory stream for text I/O. The text buffer is discarded when the `close()` method is called.

The initial value of the buffer can be set by providing `initial_value`. If newline translation is enabled, newlines will be encoded as if by `write()`. The stream is positioned at the start of the buffer.

The `newline` argument works like that of `TextIOWrapper`. The default is to consider only `\n` characters as ends of lines and to do no newline translation. If `newline` is set to `None`, newlines are written as `\n` on all platforms, but universal newline decoding is still performed when reading.

`StringIO` provides this method in addition to those from `TextIOBase` and its parents :

getvalue()

Return a `str` containing the entire contents of the buffer. Newlines are decoded as if by `read()`, although the stream position is not changed.

Exemple d'utilisation :

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class `io.IncrementalNewlineDecoder`

A helper codec that decodes newlines for *universal newlines* mode. It inherits `codecs.IncrementalDecoder`.

16.2.4 Performances

This section discusses the performance of the provided concrete I/O implementations.

Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O hides any inefficiency in calling and executing the operating system's unbuffered I/O routines. The gain depends on the OS and the kind of I/O which is performed. For example, on some modern OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O. The bottom line, however, is that buffered I/O offers predictable performance regardless of the platform and the backing device. Therefore, it is almost always preferable to use buffered I/O rather than unbuffered I/O for binary data.

Text I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it requires conversions between unicode and binary data using a character codec. This can become noticeable handling huge amounts of text data like large log files. Also, `TextIOWrapper.tell()` and `TextIOWrapper.seek()` are both quite slow due to the reconstruction algorithm used.

StringIO, however, is a native in-memory unicode container and will exhibit similar speed to *BytesIO*.

Fils d'exécution

FileIO objects are thread-safe to the extent that the operating system calls (such as `read(2)` under Unix) they wrap are thread-safe too.

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

TextIOWrapper objects are not thread-safe.

Reentrancy

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) are not reentrant. While reentrant calls will not happen in normal situations, they can arise from doing I/O in a *signal* handler. If a thread tries to re-enter a buffered object which it is already accessing, a *RuntimeError* is raised. Note this doesn't prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the *open()* function will wrap a buffered object inside a *TextIOWrapper*. This includes standard streams and therefore affects the built-in function *print()* as well.

16.3 time — Accès au temps et conversions

Ce module fournit différentes fonctions liées au temps. Pour les fonctionnalités associées, voir aussi les modules *datetime* et *calendar*.

Bien que ce module soit toujours disponible, toutes les fonctions ne sont pas disponibles sur toutes les plateformes. La plupart des fonctions définies dans ce module délèguent à des fonctions de même nom de la bibliothèque C. Il peut parfois être utile de consulter la documentation de la plate-forme, car la sémantique de ces fonctions peut varier.

Vous trouvez ci-dessous, mises en ordre, quelques explications relative à la terminologie et aux conventions.

- L'*epoch* est le point de départ du temps et dépend de la plate-forme. Pour Unix, *epoch* est le 1er janvier 1970 à 00 :00 :00 (UTC). Pour savoir comment est définie *epoch* sur une plate-forme donnée, regardez `time.gmtime(0)`.
- Le terme *secondes depuis *epoch** désigne le nombre total de secondes écoulées depuis *epoch*, souvent en excluant les secondes intercalaires (*leap seconds*). Les secondes intercalaires sont exclues de ce total sur toutes les plates-formes conformes POSIX.
- Les fonctions de ce module peuvent ne pas gérer les dates et heures antérieures à *epoch* ou dans un avenir lointain. Le seuil du futur est déterminé par la bibliothèque C; pour les systèmes 32 bits, il s'agit généralement de 2038.

- **Problèmes liés à l'an 2000 (Y2K)** : Python dépend de la bibliothèque C de la plate-forme, qui n'a généralement pas de problèmes liés à l'an 2000, étant donné que toutes les dates et heures sont représentées en interne en secondes depuis *epoch*. La fonction `strptime()` peut analyser des années à 2 chiffres lorsque le format `%Y` est spécifié. Lorsque les années à deux chiffres sont analysées, elles sont converties conformément aux normes POSIX et ISO C : les valeurs 69—99 correspondent à 1969—1999 et les valeurs 0—68 à 2000—2068.
- UTC désigne le temps universel coordonné (*Coordinated Universal Time* en anglais), anciennement l'heure de Greenwich (ou GMT). L'acronyme UTC n'est pas une erreur mais un compromis entre l'anglais et le français.
- Le DST (*Daylight Saving Time*) correspond à l'heure d'été, un ajustement du fuseau horaire d'une heure (généralement) pendant une partie de l'année. Les règles de DST sont magiques (déterminées par la loi locale) et peuvent changer d'année en année. La bibliothèque C possède une table contenant les règles locales (souvent, elle est lue dans un fichier système par souci de souplesse) et constitue la seule source fiable.
- La précision des diverses fonctions en temps réel peut être inférieure à celle suggérée par les unités dans lesquelles leur valeur ou leur argument est exprimé. Par exemple, sur la plupart des systèmes Unix, l'horloge ne « bat » que 50 ou 100 fois par seconde.
- D'autre part, la précision de `time()` et `sleep()` est meilleure que leurs équivalents Unix : les temps sont exprimés en nombres à virgule flottante, `time()` renvoie le temps le plus précis disponible (en utilisant `gettimeofday()` d'Unix si elle est disponible), et `sleep()` accepte le temps avec une fraction non nulle (`select()` d'Unix est utilisée pour l'implémenter, si elle est disponible).
- La valeur temporelle renvoyée par `gmtime()`, `localtime()` et `strptime()`, et acceptée par `asctime()`, `mktime()` et `strftime()`, est une séquence de 9 nombres entiers. Les valeurs de retour de `gmtime()`, `localtime()` et `strptime()` proposent également des noms d'attributs pour des champs individuels.

Voir `struct_time` pour une description de ces objets.

Modifié dans la version 3.3 : Le type `struct_time` a été étendu pour fournir les attributs `tm_gmtoff` et `tm_zone` lorsque la plateforme prend en charge les membres `struct tm` correspondants.

Modifié dans la version 3.6 : Les attributs `struct_time` `tm_gmtoff` et `tm_zone` sont maintenant disponibles sur toutes les plateformes.

- Utilisez les fonctions suivantes pour convertir des représentations temporelles :

De	À	Utilisez
secondes depuis <i>epoch</i>	<code>struct_time</code> en UTC	<code>gmtime()</code>
secondes depuis <i>epoch</i>	<code>struct_time</code> en heure locale	<code>localtime()</code>
<code>struct_time</code> en UTC	secondes depuis <i>epoch</i>	<code>calendar.timegm()</code>
<code>struct_time</code> en heure locale	secondes depuis <i>epoch</i>	<code>mktime()</code>

16.3.1 Fonctions

`time.asctime([t])`

Convertit un *tuple* ou `struct_time` représentant une heure renvoyée par `gmtime()` ou `localtime()` en une chaîne de la forme suivante : 'Sun Jun 20 23:21:05 1993'. Si *t* n'est pas fourni, l'heure actuelle renvoyée par `localtime()` est utilisée. Les informations sur les paramètres régionaux ne sont pas utilisées par `asctime()`.

Note : Contrairement à la fonction C du même nom, `asctime()` n'ajoute pas de caractère de fin de ligne.

`time.clock()`

Sous UNIX, renvoie le temps processeur actuel, en secondes, sous la forme d'un nombre à virgule flottante exprimé en secondes. La précision, et en fait la définition même de la signification de « temps processeur », dépend de celle de la fonction C du même nom.

Sous Windows, cette fonction renvoie les secondes réelles (type horloge murale) écoulées depuis le premier appel à cette fonction, en tant que nombre à virgule flottante, en fonction de la fonction `Win32`

`QueryPerformanceCounter()`. La résolution est généralement meilleure qu'une microseconde.

Obsolète depuis la version 3.3 : Le comportement de cette fonction dépend de la plate-forme : utilisez plutôt `perf_counter()` ou `process_time()`, selon vos besoins, pour avoir un comportement bien défini.

`time.clock_getres(clk_id)`

Renvoie la résolution (précision) de l'horloge *clk_id*. Référez-vous à *Constantes d'identification d'horloge* pour une liste des valeurs acceptées pour *clk_id*.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`time.clock_gettime(clk_id)`

Renvoie l'heure de l'horloge *clk_id*. Référez-vous à *Constantes d'identification d'horloge* pour une liste des valeurs acceptées pour *clk_id*.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`time.clock_settime(clk_id, time)`

Définit l'heure de l'horloge *clk_id*. Actuellement, `CLOCK_REALTIME` est la seule valeur acceptée pour *clk_id*.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`time.ctime([secs])`

Convertit une heure exprimée en secondes depuis *epoch* en une chaîne représentant l'heure locale. Si *secs* n'est pas fourni ou vaut `None`, l'heure actuelle renvoyée par `time()` est utilisée. `ctime(secs)` est équivalent à `asctime(localtime(secs))`. Les informations sur les paramètres régionaux ne sont pas utilisées par `ctime()`.

`time.get_clock_info(name)`

Renvoie des informations sur l'horloge spécifiée en tant qu'objet d'espace de nom. Les noms d'horloge pris en charge et les fonctions correspondantes permettant de lire leur valeur sont les suivants :

- `'clock': time.clock()`
- `'monotonic': time.monotonic()`
- `'perf_counter': time.perf_counter()`
- `'process_time': time.process_time()`
- `'time': time.time()`

Le résultat a les attributs suivants :

- `adjustable` : `True` si l'horloge peut être changée automatiquement (par exemple par un démon NTP) ou manuellement par l'administrateur système, `False` autrement
- `implementation` : nom de la fonction C sous-jacente utilisée pour obtenir la valeur d'horloge. Voir *Constantes d'identification d'horloge* pour les valeurs possibles.
- `monotonic` : `True` si l'horloge ne peut pas revenir en arrière, `False` autrement
- `resolution` : La résolution de l'horloge en secondes (*float*)

Nouveau dans la version 3.3.

`time.gmtime([secs])`

Convertit un temps exprimé en secondes depuis *epoch* en un `struct_time` au format UTC dans lequel le drapeau *dst* est toujours égal à zéro. Si *secs* n'est pas fourni ou vaut `None`, l'heure actuelle renvoyée par `time()` est utilisée. Les fractions de seconde sont ignorées. Voir ci-dessus pour une description de l'objet `struct_time`. Voir `calendar.timegm()` pour l'inverse de cette fonction.

`time.localtime([secs])`

Comme `gmtime()` mais convertit le résultat en heure locale. Si *secs* n'est pas fourni ou vaut `None`, l'heure actuelle renvoyée par `time()` est utilisée. Le drapeau *dst* est mis à 1 lorsque l'heure d'été s'applique à l'heure indiquée.

`time.mktime(t)`

C'est la fonction inverse de `localtime()`. Son argument est soit un `struct_time` soit un 9-tuple (puisque le drapeau *dst* est nécessaire ; utilisez `-1` comme drapeau *dst* s'il est inconnu) qui exprime le temps **local**, pas UTC.

Il retourne un nombre à virgule flottante, pour compatibilité avec `time()`. Si la valeur d'entrée ne peut pas être représentée comme une heure valide, soit `OverflowError` ou `ValueError` sera levée (selon que la valeur non valide est interceptée par Python ou par les bibliothèque C sous-jacentes). La date la plus proche pour laquelle il peut générer une heure dépend de la plate-forme.

`time.monotonic()`

Renvoie la valeur (en quelques fractions de secondes) d'une horloge monotone, c'est-à-dire une horloge qui ne peut pas revenir en arrière. L'horloge n'est pas affectée par les mises à jour de l'horloge système. Le point de référence de la valeur renvoyée n'est pas défini, de sorte que seule la différence entre les résultats d'appels consécutifs est valide.

On Windows versions older than Vista, `monotonic()` detects `GetTickCount()` integer overflow (32 bits, roll-over after 49.7 days). It increases an internal epoch (reference time) by 2^{32} each time that an overflow is detected. The epoch is stored in the process-local state and so the value of `monotonic()` may be different in two Python processes running for more than 49 days. On more recent versions of Windows and on other operating systems, `monotonic()` is system-wide.

Nouveau dans la version 3.3.

Modifié dans la version 3.5 : The function is now always available.

`time.perf_counter()`

Renvoie la valeur (en quelques fractions de secondes) d'un compteur de performance, c'est-à-dire une horloge avec la résolution disponible la plus élevée possible pour mesurer une courte durée. Cela inclut le temps écoulé pendant le sommeil et concerne l'ensemble du système. Le point de référence de la valeur renvoyée n'est pas défini, de sorte que seule la différence entre les résultats d'appels consécutifs est valide.

Nouveau dans la version 3.3.

`time.process_time()`

Renvoie la valeur (en quelques fractions de secondes) de la somme des temps système et utilisateur du processus en cours. Il ne comprend pas le temps écoulé pendant le sommeil. C'est un processus par définition. Le point de référence de la valeur renvoyée n'est pas défini, de sorte que seule la différence entre les résultats d'appels consécutifs est valide.

Nouveau dans la version 3.3.

`time.sleep(secs)`

Suspend l'exécution du thread appelant pendant le nombre de secondes indiqué. L'argument peut être un nombre à virgule flottante pour indiquer un temps de sommeil plus précis. Le temps de suspension réel peut être inférieur à celui demandé, car tout signal capturé mettra fin à la commande `sleep()` après l'exécution de la routine de capture de ce signal. En outre, le temps de suspension peut être plus long que celui demandé par un montant arbitraire en raison de la planification d'une autre activité dans le système.

Modifié dans la version 3.5 : La fonction dort maintenant au moins *secondes* même si le sommeil est interrompu par un signal, sauf si le gestionnaire de signaux lève une exception (voir [PEP 475](#) pour la justification).

`time.strftime(format[, t])`

Convertit un *tuple* ou `struct_time` représentant une heure renvoyée par `gmtime()` ou `localtime()` en une chaîne spécifiée par l'argument *format*. Si *t* n'est pas fourni, l'heure actuelle renvoyée par `localtime()` est utilisée. *format* doit être une chaîne. Si l'un des champs de *t* se situe en dehors de la plage autorisée, une `ValueError` est levée.

0 est un argument légal pour toute position dans le *tuple* temporel ; s'il est normalement illégal, la valeur est forcée à une valeur correcte.

Les directives suivantes peuvent être incorporées dans la chaîne *format*. Ils sont affichés sans la spécification facultative de largeur de champ ni de précision, et sont remplacés par les caractères indiqués dans le résultat de `strftime()` :

Di- rec- tive	Signification	Notes
%a	Nom abrégé du jour de la semaine selon les paramètres régionaux.	
%A	Le nom de semaine complet de la région.	
%b	Nom abrégé du mois de la région.	
%B	Nom complet du mois de la région.	
%c	Représentation appropriée de la date et de l'heure selon les paramètres régionaux.	
%d	Jour du mois sous forme décimale [01,31].	
%H	Heure (horloge sur 24 heures) sous forme de nombre décimal [00,23].	
%I	Heure (horloge sur 12 heures) sous forme de nombre décimal [01,12].	
%j	Jour de l'année sous forme de nombre décimal [001,366].	
%m	Mois sous forme décimale [01,12].	
%M	Minutes sous forme décimale [00,59].	
%p	L'équivalent local de AM ou PM.	(1)
%S	Deuxième sous forme de nombre décimal [00,61].	(2)
%U	Numéro de semaine de l'année (dimanche en tant que premier jour de la semaine) sous forme décimale [00,53]. Tous les jours d'une nouvelle année précédant le premier dimanche sont considérés comme appartenant à la semaine 0.	(3)
%w	Jour de la semaine sous forme de nombre décimal [0 (dimanche), 6].	
%W	Numéro de semaine de l'année (lundi comme premier jour de la semaine) sous forme décimale [00,53]. Tous les jours d'une nouvelle année précédant le premier lundi sont considérés comme appartenant à la semaine 0.	(3)
%x	Représentation de la date appropriée par les paramètres régionaux.	
%X	Représentation locale de l'heure.	
%y	Année sans siècle comme un nombre décimal [00, 99].	
%Y	Année complète sur quatre chiffres.	
%z	Décalage de fuseau horaire indiquant une différence de temps positive ou négative par rapport à UTC / GMT de la forme <i>+HHMM</i> ou <i>-HHMM</i> , où H représente les chiffres des heures décimales et M, les chiffres des minutes décimales [-23 :59, +23 :59].	
%Z	Nom du fuseau horaire (pas de caractères s'il n'y a pas de fuseau horaire).	
%%	Un caractère '%' littéral.	

Notes :

- (1) Lorsqu'elle est utilisée avec la fonction `strptime()`, la directive `%p` n'affecte le champ d'heure en sortie que si la directive `%I` est utilisée pour analyser l'heure.
- (2) La plage est en réalité de 0 à 61 ; La valeur 60 est valide dans les *timestamps* représentant des *leap seconds* et la valeur 61 est prise en charge pour des raisons historiques.
- (3) Lorsqu'elles sont utilisées avec la fonction `strptime()`, `%U` et `%W` ne sont utilisées que dans les calculs lorsque le jour de la semaine et l'année sont spécifiés.

Voici un exemple de format de date compatible avec celui spécifié dans la norme de courrier électronique Internet suivante [RFC 2822](#).¹

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

1. L'utilisation de `%Z` est maintenant obsolète, mais l'échappement `%z` qui donne le décalage horaire jusqu'à la minute et dépendant des paramètres régionaux n'est pas pris en charge par toutes les bibliothèques C ANSI. En outre, une lecture stricte du standard [RFC 822](#) de 1982 milite pour une année à deux chiffres (`%y` plutôt que `%Y`), mais la pratique a migré vers des années à 4 chiffres de long avant l'année 2000. Après cela, la [RFC 822](#) est devenue obsolète et l'année à 4 chiffres a été recommandée pour la première fois par la [RFC 1123](#) puis rendue obligatoire par la [RFC 2822](#).

Des directives supplémentaires peuvent être prises en charge sur certaines plates-formes, mais seules celles énumérées ici ont une signification normalisée par ANSI C. Pour voir la liste complète des codes de format pris en charge sur votre plate-forme, consultez la documentation `strptime(3)`.

Sur certaines plates-formes, une spécification facultative de largeur et de précision de champ peut suivre immédiatement le '%' initial d'une directive dans l'ordre suivant ; ce n'est pas non plus portable. La largeur du champ est normalement 2 sauf pour %j où il est 3.

`time.strptime(string[, format])`

Analyse une chaîne représentant une heure selon un format. La valeur renvoyée est une `struct_time` tel que renvoyé par `gmtime()` ou `localtime()`.

Le paramètre `format` utilise les mêmes directives que celles utilisées par `strptime()` ; La valeur par défaut est "%a %b %d %H:%M:%S %Y" qui correspond à la mise en forme renvoyée par `ctime()`. Si `string` ne peut pas être analysé selon `format`, ou s'il contient trop de données après l'analyse, une exception `ValueError` est levée. Les valeurs par défaut utilisées pour renseigner les données manquantes lorsque des valeurs plus précises ne peuvent pas être inférées sont (1900, 1, 1, 0, 0, 0, 0, 1, -1). `string` et `format` doivent être des chaînes.

Par exemple :

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                  tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

La prise en charge de la directive %Z est basée sur les valeurs contenues dans `tzname` et sur le fait de savoir si `daylight` est vrai. Pour cette raison, il est spécifique à la plate-forme, à l'exception de la reconnaissance des heures UTC et GMT, qui sont toujours connues (et considérées comme des fuseaux horaires ne respectant pas l'heure d'été).

Seules les directives spécifiées dans la documentation sont prises en charge. Parce que `strptime()` peut être implémenté différemment sur chaque plate-forme, il peut parfois offrir plus de directives que celles listées. Mais `strptime()` est indépendant de toute plate-forme et ne supporte donc pas nécessairement toutes les directives disponibles qui ne sont pas documentées comme gérées.

`class time.struct_time`

Le type de la séquence de valeur temporelle renvoyé par `gmtime()`, `localtime()` et `strptime()`. Semblable à un *named tuple* : ses valeurs sont accessibles par index et par nom d'attribut. Les valeurs suivantes sont présentes :

Index	Attribut	Valeurs
0	<code>tm_year</code>	(par exemple, 1993)
1	<code>tm_mon</code>	plage [1, 12]
2	<code>tm_mday</code>	plage [1, 31]
3	<code>tm_hour</code>	plage [0, 23]
4	<code>tm_min</code>	plage [0, 59]
5	<code>tm_sec</code>	plage [0, 61] ; voir (2) dans la description <code>strptime()</code>
6	<code>tm_wday</code>	plage [0, 6], Lundi valant 0
7	<code>tm_yday</code>	plage [1, 366]
8	<code>tm_isdst</code>	0, 1 or -1 ; voir en bas
N/A	<code>tm_zone</code>	abréviation du nom du fuseau horaire
N/A	<code>tm_gmtoff</code>	décalage à l'est de UTC en secondes

Notez que contrairement à la structure C, la valeur du mois est une plage de [1, 12], pas de [0, 11].

Dans les appels à `mktime()`, `tm_isdst` peut être défini sur 1 lorsque l'heure d'été est en vigueur et sur 0 lorsque ce n'est pas le cas. Une valeur de -1 indique que cela n'est pas connu et entraînera généralement le remplissage de l'état correct.

Lorsqu'un *tuple* de longueur incorrecte est passé à une fonction acceptant une `struct_time`, ou comportant des éléments de type incorrect, une exception `TypeError` est levé.

`time.time()`

Renvoie le temps en secondes depuis *epoch* sous forme de nombre à virgule flottante. La date spécifique de *epoch* et le traitement des secondes intercalaires (*leap seconds*) dépendent de la plate-forme. Sous Windows et la plupart des systèmes Unix, *epoch* est le 1er janvier 1970, 00 :00 :00 (UTC) et les secondes intercalaires ne sont pas comptées dans le temps en secondes depuis *epoch*. Ceci est communément appelé *Heure Unix*. Pour savoir quelle est *epoch* sur une plate-forme donnée, consultez `gmtime()`.

Notez que même si l'heure est toujours renvoyée sous forme de nombre à virgule flottante, tous les systèmes ne fournissent pas l'heure avec une précision supérieure à 1 seconde. Bien que cette fonction renvoie normalement des valeurs croissantes, elle peut renvoyer une valeur inférieure à celle d'un appel précédent si l'horloge système a été réglée entre les deux appels.

Le nombre renvoyé par `time()` peut être converti en un format d'heure plus courant (année, mois, jour, heure, etc.) en UTC en le transmettant à la fonction `gmtime()` ou dans heure locale en le transmettant à la fonction `localtime()`. Dans les deux cas, un objet `struct_time` est renvoyé, à partir duquel les composants de la date du calendrier peuvent être consultés en tant qu'attributs.

`time.tzset()`

Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.

Disponibilité : Unix.

Note : Bien que dans de nombreux cas, la modification de la variable d'environnement TZ puisse affecter la sortie de fonctions telles que `localtime()` sans appeler `tzset()`, ce comportement n'est pas garanti.

La variable d'environnement TZ ne doit contenir aucun espace.

Le format standard de la variable d'environnement TZ est (espaces ajoutés pour plus de clarté) :

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Où les composants sont :

std et dst Trois alphanumériques ou plus donnant les abréviations du fuseau horaire. Ceux-ci seront propagés dans `time.tzname`

offset Le décalage a la forme suivante : $\pm hh[:mm[:ss]]$. Cela indique la valeur ajoutée à l'heure locale pour arriver à UTC. S'il est précédé d'un '-', le fuseau horaire est à l'est du Premier Méridien ; sinon, c'est l'ouest. Si aucun décalage ne suit *dst*, l'heure d'été est supposée être en avance d'une heure sur l'heure standard.

start[/time], end[/time] Indique quand passer à DST et en revenir. Le format des dates de début et de fin est l'un des suivants :

Jn Le jour Julien *n* ($1 \leq n \leq 365$). Les jours bissextiles ne sont pas comptabilisés. Par conséquent, le 28 février est le 59^e jour et le 1^{er} Mars est le 60^e jour de toutes les années.

n Le jour Julien de base zéro ($0 \leq n \leq 365$). Les jours bissextiles sont comptés et il est possible de se référer au 29 février.

Mm.n.d Le *d* jour ($0 \leq d \leq 6$) de la semaine *n* du mois *m* de l'année ($1 \leq n \leq 5$, $1 \leq m \leq 12$, où semaine 5 signifie "le dernier jour du mois *m*" pouvant se produire au cours de la quatrième ou de la cinquième semaine). La semaine 1 est la première semaine au cours de laquelle le *jour* se produit. Le jour zéro est un dimanche.

`time` a le même format que `offset` sauf qu'aucun signe de direction ('-' ou '+') n'est autorisé. La valeur par défaut, si l'heure n'est pas spécifiée, est 02 :00 :00.

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

Sur de nombreux systèmes Unix (y compris *BSD, Linux, Solaris et Darwin), il est plus pratique d'utiliser la base de données *zoneinfo* (*tzfile* (5)) du système pour spécifier les règles de fuseau horaire. Pour ce faire, définissez la variable d'environnement TZ sur le chemin du fichier de fuseau horaire requis, par rapport à la racine de la base de données du système *zoneinfo*, généralement situé à /usr/share/zoneinfo. Par exemple, 'US/Eastern', 'Australia/Melbourne', 'Egypt' ou 'Europe/Amsterdam'.

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 Constantes d'identification d'horloge

Ces constantes sont utilisées comme paramètres pour *clock_getres()* et *clock_gettime()*.

time.CLOCK_HIGHRES

Le système d'exploitation Solaris dispose d'une horloge CLOCK_HIGHRES qui tente d'utiliser une source matérielle optimale et peut donner une résolution proche de la nanoseconde. CLOCK_HIGHRES est l'horloge haute résolution non ajustable.

Availability : Solaris.

Nouveau dans la version 3.3.

time.CLOCK_MONOTONIC

Horloge qui ne peut pas être réglée et représente l'heure monotone depuis un point de départ non spécifié.

Disponibilité : Unix.

Nouveau dans la version 3.3.

time.CLOCK_MONOTONIC_RAW

Similaire à *CLOCK_MONOTONIC*, mais donne accès à une heure matérielle brute qui n'est pas soumise aux ajustements NTP.

Availability : Linux 2.6.28 or later.

Nouveau dans la version 3.3.

time.CLOCK_PROCESS_CPUTIME_ID

Minuterie haute résolution par processus du CPU.

Disponibilité : Unix.

Nouveau dans la version 3.3.

time.CLOCK_THREAD_CPUTIME_ID

Horloge de temps CPU spécifique au thread.

Disponibilité : Unix.

Nouveau dans la version 3.3.

La constante suivante est le seul paramètre pouvant être envoyé à *clock_settime()*.

time.CLOCK_REALTIME

Horloge en temps réel à l'échelle du système. Le réglage de cette horloge nécessite des privilèges appropriés.

Disponibilité : Unix.

Nouveau dans la version 3.3.

16.3.3 Constantes de fuseau horaire

`time.altzone`

Décalage du fuseau horaire DST local, en secondes à l'ouest de UTC, s'il en est défini un. Cela est négatif si le fuseau horaire DST local est à l'est de UTC (comme en Europe occidentale, y compris le Royaume-Uni). Utilisez ceci uniquement si `daylight` est différent de zéro. Voir note ci-dessous.

`time.daylight`

Non nul si un fuseau horaire DST est défini. Voir note ci-dessous.

`time.timezone`

Décalage du fuseau horaire local (hors heure d'été), en secondes à l'ouest de l'UTC (négatif dans la plupart des pays d'Europe occidentale, positif aux États-Unis, nul au Royaume-Uni). Voir note ci-dessous.

`time.tzname`

Un *tuple* de deux chaînes : la première est le nom du fuseau horaire local autre que DST, la seconde est le nom du fuseau horaire DST local. Si aucun fuseau horaire DST n'est défini, la deuxième chaîne ne doit pas être utilisée. Voir note ci-dessous.

Note : Pour les constantes de fuseau horaire ci-dessus (`altzone`, `daylight`, `timezone` et `tzname`), la valeur est déterminée par les règles de fuseau horaire en vigueur au moment du chargement du module ou la dernière fois `tzset()` est appelé et peut être incorrect pour des temps passés. Il est recommandé d'utiliser `tm_gmtoff` et `tm_zone` résulte de `localtime()` pour obtenir des informations sur le fuseau horaire.

Voir aussi :

Module `datetime` Interface plus orientée objet vers les dates et les heures.

Module `locale` Services d'internationalisation. Les paramètres régionaux affectent l'interprétation de nombreux spécificateurs de format dans `strftime()` et `strptime()`.

Module `calendar` Fonctions générales liées au calendrier. `timegm()` est l'inverse de `gmtime()` à partir de ce module.

Notes

16.4 argparse – Parseur d'arguments, d'options, et de sous-commandes de ligne de commande

Nouveau dans la version 3.2.

Code source : [Lib/argparse.py](#)

Tutoriel

Cette page est la référence de l'API. Pour une introduction plus en douceur à l'analyse des arguments de la ligne de commande, regardez le tutoriel `argparse`.

Le module `argparse` facilite l'écriture d'interfaces en ligne de commande agréables à l'emploi. Le programme définit les arguments requis et `argparse` s'arrange pour analyser ceux provenant de `sys.argv`. Le module `argparse` génère aussi automatiquement les messages d'aide, le mode d'emploi, et lève des erreurs lorsque les utilisateurs fournissent au programme des arguments invalides.

16.4.1 Exemple

Le code suivant est un programme Python acceptant une liste de nombre entiers et en donnant soit la somme, soit le maximum :

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

En supposant que le code Python ci-dessus est sauvegardé dans un fichier nommé `prog.py`, il peut être lancé en ligne de commande et fournit des messages d'aide utiles :

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
  N              an integer for the accumulator

optional arguments:
  -h, --help    show this help message and exit
  --sum         sum the integers (default: find the max)
```

Lorsqu'il est lancé avec les arguments appropriés, il affiche la somme ou le maximum des entiers fournis en ligne de commande :

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

Si des arguments invalides sont passés, il lève une erreur :

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

Les sections suivantes vous guident au travers de cet exemple.

Créer un analyseur (*parser* en anglais)

La première étape dans l'utilisation de `argparse` est de créer un objet `ArgumentParser` :

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

L'objet `ArgumentParser` contiendra toutes les informations nécessaires pour interpréter la ligne de commande comme des types de données de Python.

Ajouter des arguments

Alimenter un `ArgumentParser` avec des informations sur les arguments du programme s'effectue en faisant des appels à la méthode `add_argument()`. En général ces appels disent à l'`ArgumentParser` comment prendre les chaînes de caractères de la ligne de commande et les transformer en objets. Cette information est stockée et utilisée lorsque `parse_args()` est appelée. Par exemple :

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

Ensuite, appeler `parse_args()` va renvoyer un objet avec deux attributs, `integers` et `accumulate`. L'attribut `integers` est une liste d'un ou plusieurs entiers, et l'attribut `accumulate` est soit la fonction `sum()`, si `--sum` était fourni à la ligne de commande, soit la fonction `max()` dans le cas contraire.

Analyse des arguments

`ArgumentParser` parses arguments through the `parse_args()` method. This will inspect the command line, convert each argument to the appropriate type and then invoke the appropriate action. In most cases, this means a simple `Namespace` object will be built up from attributes parsed out of the command line :

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

In a script, `parse_args()` will typically be called with no arguments, and the `ArgumentParser` will automatically determine the command-line arguments from `sys.argv`.

16.4.2 Objets `ArgumentParser`

```
class argparse.ArgumentParser (prog=None, usage=None, description=None, epilog=None, pa-
                             rents=[], formatter_class=argparse.HelpFormatter, prefix_chars='-'
                             , fromfile_prefix_chars=None, argument_default=None,
                             conflict_handler='error', add_help=True, allow_abbrev=True)
```

Create a new `ArgumentParser` object. All parameters should be passed as keyword arguments. Each parameter has its own more detailed description below, but in short they are :

- `prog` - The name of the program (default : `sys.argv[0]`)
- `usage` - The string describing the program usage (default : generated from arguments added to parser)
- `description` - Text to display before the argument help (default : none)
- `epilog` - Text to display after the argument help (default : none)
- `parents` - A list of `ArgumentParser` objects whose arguments should also be included
- `formatter_class` - A class for customizing the help output
- `prefix_chars` - The set of characters that prefix optional arguments (default : “-“)

- *fromfile_prefix_chars* - The set of characters that prefix files from which additional arguments should be read (default : None)
- *argument_default* - The global default value for arguments (default : None)
- *conflict_handler* - The strategy for resolving conflicting optionals (usually unnecessary)
- *add_help* - Add a `-h/--help` option to the parser (default : True)
- *allow_abbrev* - Allows long options to be abbreviated if the abbreviation is unambiguous. (default : True)

Modifié dans la version 3.5 : Le paramètre *allow_abbrev* est ajouté.

The following sections describe how each of these are used.

Le paramètre *prog*

By default, *ArgumentParser* objects use `sys.argv[0]` to determine how to display the name of the program in help messages. This default is almost always desirable because it will make the help messages match how the program was invoked on the command line. For example, consider a file named `myprogram.py` with the following code :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

The help for this program will display `myprogram.py` as the program name (regardless of where the program was invoked from) :

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo help
```

To change this default behavior, another value can be supplied using the *prog=* argument to *ArgumentParser* :

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
  -h, --help  show this help message and exit
```

Note that the program name, whether determined from `sys.argv[0]` or from the *prog=* argument, is available to help messages using the `%(prog)s` format specifier.

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
```

(suite sur la page suivante)

(suite de la page précédente)

```
-h, --help  show this help message and exit
--foo FOO   foo of the myprogram program
```

Le paramètre *usage*

By default, `ArgumentParser` calculates the usage message from the arguments it contains :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
  bar                bar help

optional arguments:
  -h, --help  show this help message and exit
  --foo [FOO] foo help
```

The default message can be overridden with the `usage=` keyword argument :

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='% (prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
  bar                bar help

optional arguments:
  -h, --help  show this help message and exit
  --foo [FOO] foo help
```

The `% (prog)s` format specifier is available to fill in the program name in your usage messages.

Le paramètre *description*

Most calls to the `ArgumentParser` constructor will use the `description=` keyword argument. This argument gives a brief description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments :

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
  -h, --help  show this help message and exit
```

By default, the description will be line-wrapped so that it fits within the given space. To change this behavior, see the `formatter_class` argument.

Le paramètre *epilog*

Some programs like to display additional description of the program after the description of the arguments. Such text can be specified using the `epilog=` argument to `ArgumentParser`:

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
  -h, --help  show this help message and exit

And that's how you'd foo a bar
```

As with the `description` argument, the `epilog=` text is by default line-wrapped, but this behavior can be adjusted with the `formatter_class` argument to `ArgumentParser`.

Le paramètre *parents*

Sometimes, several parsers share a common set of arguments. Rather than repeating the definitions of these arguments, a single parser with all the shared arguments and passed to `parents=` argument to `ArgumentParser` can be used. The `parents=` argument takes a list of `ArgumentParser` objects, collects all the positional and optional actions from them, and adds these actions to the `ArgumentParser` object being constructed:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

Note that most parent parsers will specify `add_help=False`. Otherwise, the `ArgumentParser` will see two `-h/` `--help` options (one in the parent and one in the child) and raise an error.

Note : You must fully initialize the parsers before passing them via `parents=`. If you change the parent parsers after the child parser, those changes will not be reflected in the child.

Le paramètre `formatter_class`

`ArgumentParser` objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are four such classes :

```
class argparse.RawDescriptionHelpFormatter
class argparse.RawTextHelpFormatter
class argparse.ArgumentDefaultsHelpFormatter
class argparse.MetavarTypeHelpFormatter
```

`RawDescriptionHelpFormatter` and `RawTextHelpFormatter` give more control over how textual descriptions are displayed. By default, `ArgumentParser` objects line-wrap the *description* and *epilog* texts in command-line help messages :

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...         but that is okay''',
...     epilog='''
...         likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

optional arguments:
  -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

Passing `RawDescriptionHelpFormatter` as `formatter_class=` indicates that *description* and *epilog* are already correctly formatted and should not be line-wrapped :

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         -----
...         I have indented it
...         exactly the way
...         I want it
...     '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
-----

    I have indented it
    exactly the way
    I want it

optional arguments:
  -h, --help  show this help message and exit
```

RawTextHelpFormatter maintains whitespace for all sorts of help text, including argument descriptions. However, multiple new lines are replaced with one. If you wish to preserve multiple blank lines, add spaces between the newlines.

ArgumentDefaultsHelpFormatter automatically adds information about default values to each of the argument help messages :

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
  bar          BAR! (default: [1, 2, 3])

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter uses the name of the *type* argument for each argument as the display name for its values (rather than using the *dest* as the regular formatter does) :

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  -h, --help  show this help message and exit
  --foo int
```

Le paramètre *prefix_chars*

Most command-line options will use `-` as the prefix, e.g. `-f`/`--foo`. Parsers that need to support different or additional prefix characters, e.g. for options like `+f` or `/foo`, may specify them using the `prefix_chars=` argument to the `ArgumentParser` constructor :

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+-')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

The `prefix_chars=` argument defaults to `'-'`. Supplying a set of characters that does not include `-` will cause `-f`/`--foo` options to be disallowed.

Le paramètre `fromfile_prefix_chars`

Sometimes, for example when dealing with a particularly long argument lists, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the `fromfile_prefix_chars=` argument is given to the `ArgumentParser` constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example :

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

Arguments read from a file must by default be one per line (but see also `convert_arg_line_to_args()`) and are treated as if they were in the same place as the original file referencing argument on the command line. So in the example above, the expression `['-f', 'foo', '@args.txt']` is considered equivalent to the expression `['-f', 'foo', '-f', 'bar']`.

The `fromfile_prefix_chars=` argument defaults to `None`, meaning that arguments will never be treated as file references.

Le paramètre `argument_default`

Generally, argument defaults are specified either by passing a default to `add_argument()` or by calling the `set_defaults()` methods with a specific set of name-value pairs. Sometimes however, it may be useful to specify a single parser-wide default for arguments. This can be accomplished by passing the `argument_default=` keyword argument to `ArgumentParser`. For example, to globally suppress attribute creation on `parse_args()` calls, we supply `argument_default=SUPPRESS` :

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

Le paramètre `allow_abbrev`

Normally, when you pass an argument list to the `parse_args()` method of an `ArgumentParser`, it recognizes *abbreviations* of long options.

This feature can be disabled by setting `allow_abbrev` to `False` :

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

Nouveau dans la version 3.5.

Le paramètre *conflict_handler*

ArgumentParser objects do not allow two actions with the same option string. By default, *ArgumentParser* objects raise an exception if an attempt is made to create an argument with an option string that is already in use :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

Sometimes (e.g. when using *parents*) it may be useful to simply override any older arguments with the same option string. To get this behavior, the value 'resolve' can be supplied to the *conflict_handler*= argument of *ArgumentParser* :

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  -f FOO      old foo help
  --foo FOO   new foo help
```

Note that *ArgumentParser* objects only remove an action if all of its option strings are overridden. So, in the example above, the old `-f/--foo` action is retained as the `-f` action, because only the `--foo` option string was overridden.

Le paramètre *add_help*

By default, *ArgumentParser* objects add an option which simply displays the parser's help message. For example, consider a file named `myprogram.py` containing the following code :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

If `-h` or `--help` is supplied at the command line, the *ArgumentParser* help will be printed :

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo help
```

Occasionally, it may be useful to disable the addition of this help option. This can be achieved by passing `False` as the *add_help*= argument to *ArgumentParser* :

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]
```

(suite sur la page suivante)

(suite de la page précédente)

```
optional arguments:
  --foo FOO  foo help
```

The help option is typically `-h/--help`. The exception to this is if the `prefix_chars=` is specified and does not include `-`, in which case `-h` and `--help` are not valid options. In this case, the first character in `prefix_chars` is used to prefix the help options :

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

16.4.3 La méthode `add_argument()`

`ArgumentParser.add_argument` (*name or flags*...[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*])

Define how a single command-line argument should be parsed. Each parameter has its own more detailed description below, but in short they are :

- *name or flags* - Either a name or a list of option strings, e.g. `foo` or `-f`, `--foo`.
- *action* - The basic type of action to be taken when this argument is encountered at the command line.
- *nargs* - The number of command-line arguments that should be consumed.
- *const* - A constant value required by some *action* and *nargs* selections.
- *default* - The value produced if the argument is absent from the command line.
- *type* - The type to which the command-line argument should be converted.
- *choices* - A container of the allowable values for the argument.
- *required* - Whether or not the command-line option may be omitted (optionals only).
- *help* - A brief description of what the argument does.
- *metavar* - A name for the argument in usage messages.
- *dest* - The name of the attribute to be added to the object returned by `parse_args()`.

The following sections describe how each of these are used.

Les paramètres *name* et *flags*

The `add_argument()` method must know whether an optional argument, like `-f` or `--foo`, or a positional argument, like a list of filenames, is expected. The first arguments passed to `add_argument()` must therefore be either a series of flags, or a simple argument name. For example, an optional argument could be created like :

```
>>> parser.add_argument('-f', '--foo')
```

while a positional argument could be created like :

```
>>> parser.add_argument('bar')
```

When `parse_args()` is called, optional arguments will be identified by the `-` prefix, and the remaining arguments will be assumed to be positional :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

Le paramètre *action*

ArgumentParser objects associate command-line arguments with actions. These actions can do just about anything with the command-line arguments associated with them, though most actions simply add an attribute to the object returned by *parse_args()*. The *action* keyword argument specifies how the command-line arguments should be handled. The supplied actions are :

- 'store' - This just stores the argument's value. This is the default action. For example :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args('--foo 1'.split())
Namespace(foo='1')
```

- 'store_const' - This stores the value specified by the *const* keyword argument. The 'store_const' action is most commonly used with optional arguments that specify some sort of flag. For example :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- 'store_true' and 'store_false' - These are special cases of 'store_const' used for storing the values True and False respectively. In addition, they create default values of False and True respectively. For example :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- 'append' - This stores a list, and appends each argument value to the list. This is useful to allow an option to be specified multiple times. Example usage :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- 'append_const' - This stores a list, and appends the value specified by the *const* keyword argument to the list. (Note that the *const* keyword argument defaults to None.) The 'append_const' action is typically useful when multiple arguments need to store constants to the same list. For example :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- 'count' - This counts the number of times a keyword argument occurs. For example, this is useful for increasing verbosity levels :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count')
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

- 'help' - This prints a complete help message for all the options in the current parser and then exits. By default a help action is automatically added to the parser. See [ArgumentParser](#) for details of how the output is created.
- 'version' - This expects a version= keyword argument in the [add_argument\(\)](#) call, and prints version information and exits when invoked :

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='% (prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

You may also specify an arbitrary action by passing an Action subclass or other object that implements the same interface. The recommended way to do this is to extend [Action](#), overriding the `__call__` method and optionally the `__init__` method.

Un exemple d'action personnalisée :

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super(FooAction, self).__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

Pour plus d'information, voir [Action](#).

Le paramètre *nargs*

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. The *nargs* keyword argument associates a different number of command-line arguments with a single action. The supported values are :

- *N* (an integer). *N* arguments from the command line will be gathered together into a list. For example :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs=2)
>>> parser.add_argument('bar', nargs=1)
>>> parser.parse_args('c --foo a b'.split())
Namespace(bar=['c'], foo=['a', 'b'])
```

Note that *nargs=1* produces a list of one item. This is different from the default, in which the item is produced by itself.

- *'?'*. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from *default* will be produced. Note that for optional arguments, there is an additional case - the option string is present but not followed by a command-line argument. In this case the value from *const* will be produced. Some examples to illustrate this :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='?', const='c', default='d')
>>> parser.add_argument('bar', nargs='?', default='d')
>>> parser.parse_args(['XX', '--foo', 'YY'])
Namespace(bar='XX', foo='YY')
>>> parser.parse_args(['XX', '--foo'])
Namespace(bar='XX', foo='c')
>>> parser.parse_args([])
Namespace(bar='d', foo='d')
```

One of the more common uses of *nargs='?'* is to allow optional input and output files :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
...                     default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
...                     default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
```

- *'*'*. All command-line arguments present are gathered into a list. Note that it generally doesn't make much sense to have more than one positional argument with *nargs='*'*, but multiple optional arguments with *nargs='*'* is possible. For example :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- *'+'*. Just like *'*'*, all command-line args present are gathered into a list. Additionally, an error message will be generated if there wasn't at least one command-line argument present. For example :


```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

- `argparse.REMAINDER`. All the remaining command-line arguments are gathered into a list. This is commonly useful for command line utilities that dispatch to other command line utilities :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print(parser.parse_args('--foo B cmd --arg1 XX ZZ'.split()))
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')
```

If the `nargs` keyword argument is not provided, the number of arguments consumed is determined by the *action*. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced.

Le paramètre *const*

The `const` argument of `add_argument()` is used to hold constant values that are not read from the command line but are required for the various `ArgumentParser` actions. The two most common uses of it are :

- When `add_argument()` is called with `action='store_const'` or `action='append_const'`. These actions add the `const` value to one of the attributes of the object returned by `parse_args()`. See the *action* description for examples.
- When `add_argument()` is called with option strings (like `-f` or `--foo`) and `nargs='?'`. This creates an optional argument that can be followed by zero or one command-line arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value of `const` will be assumed instead. See the *nargs* description for examples.

With the `'store_const'` and `'append_const'` actions, the `const` keyword argument must be given. For other actions, it defaults to `None`.

Le paramètre *default*

All optional arguments and some positional arguments may be omitted at the command line. The `default` keyword argument of `add_argument()`, whose value defaults to `None`, specifies what value should be used if the command-line argument is not present. For optional arguments, the `default` value is used when the option string was not present at the command line :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

If the `default` value is a string, the parser parses the value as if it were a command-line argument. In particular, the parser applies any *type* conversion argument, if provided, before setting the attribute on the `Namespace` return value. Otherwise, the parser uses the value as is :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

For positional arguments with *nargs* equal to `?` or `*`, the default value is used when no command-line argument was present :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

Providing `default=argparse.SUPPRESS` causes no attribute to be added if the command-line argument was not present :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

Le paramètre *type*

By default, *ArgumentParser* objects read command-line arguments in as simple strings. However, quite often the command-line string should instead be interpreted as another type, like a *float* or *int*. The *type* keyword argument of *add_argument()* allows any necessary type-checking and type conversions to be performed. Common built-in types and functions can be used directly as the value of the *type* argument :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=open)
>>> parser.parse_args('2 temp.txt'.split())
Namespace(bar=<_io.TextIOWrapper name='temp.txt' encoding='UTF-8'>, foo=2)
```

See the section on the *default* keyword argument for information on when the *type* argument is applied to default arguments.

To ease the use of various types of files, the *argparse* module provides the factory *FileType* which takes the *mode=*, *bufsize=*, *encoding=* and *errors=* arguments of the *open()* function. For example, *FileType('w')* can be used to create a writable file :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<_io.TextIOWrapper name='out.txt' encoding='UTF-8'>)
```

type= can take any callable that takes a single string argument and returns the converted value :

```
>>> def perfect_square(string):
...     value = int(string)
```

(suite sur la page suivante)

(suite de la page précédente)

```

...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = "%r is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
...
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args(['9'])
Namespace(foo=9)
>>> parser.parse_args(['7'])
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square

```

The *choices* keyword argument may be more convenient for type checkers that simply check against a range of values :

```

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=range(5, 10))
>>> parser.parse_args(['7'])
Namespace(foo=7)
>>> parser.parse_args(['11'])
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)

```

Voir le chapitre *choices* pour plus de détails.

Le paramètre *choices*

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a container object as the *choices* keyword argument to *add_argument()*. When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values :

```

>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')

```

Note that inclusion in the *choices* container is checked after any *type* conversions have been performed, so the type of the objects in the *choices* container should match the *type* specified :

```

>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)

```

Any object that supports the *in* operator can be passed as the *choices* value, so *dict* objects, *set* objects, custom containers, etc. are all supported.

Le paramètre *required*

In general, the `argparse` module assumes that flags like `-f` and `--bar` indicate *optional* arguments, which can always be omitted at the command line. To make an option *required*, `True` can be specified for the `required=` keyword argument to `add_argument()` :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: argparse.py [-h] [--foo FOO]
argparse.py: error: option --foo is required
```

As the example shows, if an option is marked as `required`, `parse_args()` will report an error if that option is not present at the command line.

Note : Required options are generally considered bad form because users expect *options* to be *optional*, and thus they should be avoided when possible.

Le paramètre *help*

The `help` value is a string containing a brief description of the argument. When a user requests help (usually by using `-h` or `--help` at the command line), these help descriptions will be displayed with each argument :

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
  bar                one of the bars to be frobbled

optional arguments:
  -h, --help        show this help message and exit
  --foo             foo the bars before frobbling
```

The `help` strings can include various format specifiers to avoid repetition of things like the program name or the argument *default*. The available specifiers include the program name, `%(prog)s` and most keyword arguments to `add_argument()`, e.g. `%(default)s`, `%(type)s`, etc. :

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
  bar                the bar to frobble (default: 42)

optional arguments:
  -h, --help        show this help message and exit
```

As the help string supports %-formatting, if you want a literal % to appear in the help string, you must escape it as %%.

`argparse` supports silencing the help entry for certain options, by setting the help value to `argparse.SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

Le paramètre *metavar*

When `ArgumentParser` generates help messages, it needs some way to refer to each expected argument. By default, `ArgumentParser` objects use the *dest* value as the « name » of each object. By default, for positional argument actions, the *dest* value is used directly, and for optional argument actions, the *dest* value is uppercased. So, a single positional argument with *dest*='bar' will be referred to as bar. A single optional argument `--foo` that should be followed by a single command-line argument will be referred to as FOO. An example :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage: [-h] [--foo FOO] bar

positional arguments:
  bar

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO
```

An alternative name can be specified with *metavar* :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage: [-h] [--foo YYY] XXX

positional arguments:
  XXX

optional arguments:
  -h, --help  show this help message and exit
  --foo YYY
```

Note that *metavar* only changes the *displayed* name - the name of the attribute on the `parse_args()` object is still determined by the *dest* value.

Different values of *nargs* may cause the *metavar* to be used multiple times. Providing a tuple to *metavar* specifies a different display for each of the arguments :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
  -h, --help            show this help message and exit
  -x X X
  --foo bar baz
```

Le paramètre *dest*

Most `ArgumentParser` actions add some value as an attribute of the object returned by `parse_args()`. The name of this attribute is determined by the `dest` keyword argument of `add_argument()`. For positional argument actions, `dest` is normally supplied as the first argument to `add_argument()` :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

For optional argument actions, the value of `dest` is normally inferred from the option strings. `ArgumentParser` generates the value of `dest` by taking the first long option string and stripping away the initial `--` string. If no long option strings were supplied, `dest` will be derived from the first short option string by stripping the initial `-` character. Any internal `-` characters will be converted to `_` characters to make sure the string is a valid attribute name. The examples below illustrate this behavior :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

`dest` allows a custom attribute name to be provided :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

Classes Action

Action classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the `action` parameter to `add_argument()`.

class `argparse.Action`(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Action objects are used by an `ArgumentParser` to represent the information needed to parse a single argument from one or more strings from the command line. The Action class must accept the two positional arguments plus any keyword arguments passed to `ArgumentParser.add_argument()` except for the `action` itself.

Instances of `Action` (or return value of any callable to the `action` parameter) should have attributes « `dest` », « `option_strings` », « `default` », « `type` », « `required` », « `help` », etc. defined. The easiest way to ensure these attributes are defined is to call `Action.__init__`.

Action instances should be callable, so subclasses must override the `__call__` method, which should accept four parameters :

- `parser` - The `ArgumentParser` object which contains this action.
- `namespace` - The `Namespace` object that will be returned by `parse_args()`. Most actions add an attribute to this object using `setattr()`.
- `values` - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the `type` keyword argument to `add_argument()`.
- `option_string` - The option string that was used to invoke this action. The `option_string` argument is optional, and will be absent if the action is associated with a positional argument.

The `__call__` method may perform arbitrary actions, but will typically set attributes on the `namespace` based on `dest` and `values`.

16.4.4 La méthode `parse_args()`

`ArgumentParser.parse_args(args=None, namespace=None)`

Convert argument strings to objects and assign them as attributes of the namespace. Return the populated namespace.

Previous calls to `add_argument()` determine exactly what objects are created and how they are assigned. See the documentation for `add_argument()` for details.

- `args` - List of strings to parse. The default is taken from `sys.argv`.
- `namespace` - An object to take the attributes. The default is a new empty `Namespace` object.

Option value syntax

The `parse_args()` method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), the option and value can also be passed as a single command-line argument, using `=` to separate them :

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

For short options (options only one character long), the option and its value can be concatenated :

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

Several short options can be joined together, using only a single `-` prefix, as long as only the last option (or none of them) requires a value :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

Arguments invalides

While parsing the command line, `parse_args()` checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

Arguments contenant –

The `parse_args()` method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line argument `-1` could either be an attempt to specify an option or an attempt to provide a positional argument. The `parse_args()` method is cautious here : positional arguments may only begin with `-` if they look like negative numbers and there are no options in the parser that look like negative numbers :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

If you have positional arguments that must begin with `-` and don't look like negative numbers, you can insert the pseudo-argument `--` which tells `parse_args()` that everything after that is a positional argument :

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

Arguments abrégés (Par comparaison de leurs préfixes)

The `parse_args()` method *by default* allows long options to be abbreviated to a prefix, if the abbreviation is unambiguous (the prefix matches a unique option) :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args(['-bac MMM'].split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args(['-bad WOOD'].split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args(['-ba BA'].split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

An error is produced for arguments that could produce more than one options. This feature can be disabled by setting *Le paramètre* `allow_abbrev` to `False`.

Au-delà de `sys.argv`

Sometimes it may be useful to have an `ArgumentParser` parse arguments other than those of `sys.argv`. This can be accomplished by passing a list of strings to `parse_args()`. This is useful for testing at the interactive prompt :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

L'objet namespace

`class` `argparse.Namespace`

Simple class used by default by `parse_args()` to create an object holding attributes and return it.

This class is deliberately simple, just an *object* subclass with a readable string representation. If you prefer to have dict-like view of the attributes, you can use the standard Python idiom, `vars()` :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

It may also be useful to have an *ArgumentParser* assign attributes to an already existing object, rather than a new *Namespace* object. This can be achieved by specifying the `namespace=` keyword argument :

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

16.4.5 Autres outils

Sous commandes

`ArgumentParser.add_subparsers` (`[title]`, `[description]`, `[prog]`, `[parser_class]`, `[action]`, `[option_string]`, `[dest]`, `[help]`, `[metavar]`)

Many programs split up their functionality into a number of sub-commands, for example, the `svn` program can invoke sub-commands like `svn checkout`, `svn update`, and `svn commit`. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. *ArgumentParser* supports the creation of such sub-commands with the `add_subparsers()` method. The `add_subparsers()` method is normally called with no arguments and returns a special action object. This object has a single method, `add_parser()`, which takes a command name and any *ArgumentParser* constructor arguments, and returns an *ArgumentParser* object that can be modified as usual.

Description des paramètres

- `title` - title for the sub-parser group in help output; by default « subcommands » if description is provided, otherwise uses title for positional arguments
- `description` - description for the sub-parser group in help output, by default `None`
- `prog` - usage information that will be displayed with sub-command help, by default the name of the program and any positional arguments before the subparser argument

- `parser_class` - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. `ArgumentParser`)
- `action` - the basic type of action to be taken when this argument is encountered at the command line
- `dest` - name of the attribute under which sub-command name will be stored; by default `None` and no value is stored
- `help` - help for sub-parser group in help output, by default `None`
- `metavar` - string presenting available sub-commands in help; by default it is `None` and presents sub-commands in form `{cmd1, cmd2, ...}`

Quelques exemples d'utilisation :

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Note that the object returned by `parse_args()` will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the `a` command is specified, only the `foo` and `bar` attributes are present, and when the `b` command is specified, only the `foo` and `baz` attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the `help=` argument to `add_parser()` as above.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}  sub-command help
  a      a help
  b      b help

optional arguments:
  -h, --help  show this help message and exit
  --foo      foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar      bar help

optional arguments:
  -h, --help  show this help message and exit
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help      show this help message and exit
  --baz {X,Y,Z}   baz help
```

The `add_subparsers()` method also supports `title` and `description` keyword arguments. When either is present, the subparser's commands will appear in their own group in the help output. For example :

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                   description='valid subcommands',
...                                   help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage: [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

Furthermore, `add_parser` supports an additional `aliases` argument, which allows multiple strings to refer to the same subparser. This example, like `svn`, aliases `co` as a shorthand for `checkout` :

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

One particularly effective way of handling sub-commands is to combine the use of the `add_subparsers()` method with calls to `set_defaults()` so that each subparser knows which Python function it should execute. For example :

```
>>> # sub-command functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

This way, you can let `parse_args()` do the job of calling the appropriate function after argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if it is necessary to check the name of the subparser that was invoked, the `dest` keyword argument to the `add_subparsers()` call will work :

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

Objets `FileType`

class `argparse.FileType` (*mode='r', bufsize=-1, encoding=None, errors=None*)

The `FileType` factory creates objects that can be passed to the `type` argument of `ArgumentParser.add_argument()`. Arguments that have `FileType` objects as their type will open command-line arguments as files with the requested modes, buffer sizes, encodings and error handling (see the `open()` function for more details) :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=
↳<_io.FileIO name='raw.dat' mode='wb'>)
```

`FileType` objects understand the pseudo-argument `'-'` and automatically convert this into `sys.stdin` for readable `FileType` objects and `sys.stdout` for writable `FileType` objects :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

Nouveau dans la version 3.4 : Les arguments nommés `encodings` et `errors`.

Groupes d'arguments

`ArgumentParser.add_argument_group` (*title=None, description=None*)

By default, `ArgumentParser` groups command-line arguments into « positional arguments » and « optional arguments » when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the `add_argument_group()` method :

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar      bar help
  --foo FOO  foo help
```

The `add_argument_group()` method returns an argument group object which has an `add_argument()` method just like a regular `ArgumentParser`. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The `add_argument_group()` method accepts *title* and *description* arguments which can be used to customize this display :

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo      foo help

group2:
  group2 description

  --bar BAR  bar help
```

Note that any arguments not in your user-defined groups will end up back in the usual « positional arguments » and « optional arguments » sections.

Exclusion mutuelle

`ArgumentParser.add_mutually_exclusive_group` (*required=False*)

Create a mutually exclusive group. `argparse` will make sure that only one of the arguments in the mutually exclusive group was present on the command line :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
```

(suite sur la page suivante)

(suite de la page précédente)

```

Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo

```

The `add_mutually_exclusive_group()` method also accepts a *required* argument, to indicate that at least one of the mutually exclusive arguments is required :

```

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required

```

Note that currently mutually exclusive argument groups do not support the *title* and *description* arguments of `add_argument_group()`.

Valeurs par défaut de l'analyseur

`ArgumentParser.set_defaults(**kwargs)`

Most of the time, the attributes of the object returned by `parse_args()` will be fully determined by inspecting the command-line arguments and the argument actions. `set_defaults()` allows some additional attributes that are determined without any inspection of the command line to be added :

```

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)

```

Note that parser-level defaults always override argument-level defaults :

```

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')

```

Parser-level defaults can be particularly useful when working with multiple parsers. See the `add_subparsers()` method for an example of this type.

`ArgumentParser.get_default(dest)`

Get the default value for a namespace attribute, as set by either `add_argument()` or by `set_defaults()` :

```

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'

```

Afficher l'aide

In most typical applications, `parse_args()` will take care of formatting and printing any usage or error messages. However, several formatting methods are available :

`ArgumentParser.print_usage(file=None)`

Print a brief description of how the `ArgumentParser` should be invoked on the command line. If `file` is `None`, `sys.stdout` is assumed.

`ArgumentParser.print_help(file=None)`

Print a help message, including the program usage and information about the arguments registered with the `ArgumentParser`. If `file` is `None`, `sys.stdout` is assumed.

There are also variants of these methods that simply return a string instead of printing it :

`ArgumentParser.format_usage()`

Return a string containing a brief description of how the `ArgumentParser` should be invoked on the command line.

`ArgumentParser.format_help()`

Return a string containing a help message, including the program usage and information about the arguments registered with the `ArgumentParser`.

Parsing partiel

`ArgumentParser.parse_known_args(args=None, namespace=None)`

Sometimes a script may only parse a few of the command-line arguments, passing the remaining arguments on to another script or program. In these cases, the `parse_known_args()` method can be useful. It works much like `parse_args()` except that it does not produce an error when extra arguments are present. Instead, it returns a two item tuple containing the populated namespace and the list of remaining argument strings.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

Avertissement : *Prefix matching* rules apply to `parse_known_args()`. The parser may consume an option even if it's just a prefix of one of its known options, instead of leaving it in the remaining arguments list.

Personnaliser le parsing de fichiers

`ArgumentParser.convert_arg_line_to_args(arg_line)`

Arguments that are read from a file (see the `fromfile_prefix_chars` keyword argument to the `ArgumentParser` constructor) are read one argument per line. `convert_arg_line_to_args()` can be overridden for fancier reading.

This method takes a single argument `arg_line` which is a string read from the argument file. It returns a list of arguments parsed from this string. The method is called once per line read from the argument file, in order.

A useful override of this method is one that treats each space-separated word as an argument. The following example demonstrates how to do this :


```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

Exiting methods

`ArgumentParser.exit(status=0, message=None)`

This method terminates the program, exiting with the specified *status* and, if given, it prints a *message* before that.

`ArgumentParser.error(message)`

This method prints a usage message including the *message* to the standard error and terminates the program with a status code of 2.

16.4.6 Mettre à jour du code `optparse`

Originally, the `argparse` module had attempted to maintain compatibility with `optparse`. However, `optparse` was difficult to extend transparently, particularly with the changes required to support the new `nargs=` specifiers and better usage messages. When most everything in `optparse` had either been copy-pasted over or monkey-patched, it no longer seemed practical to try to maintain the backwards compatibility.

The `argparse` module improves on the standard library `optparse` module in a number of ways including :

- Gère les arguments positionnels.
- Prise en charge des sous commandes.
- Permet d'utiliser les alternatives + ou / comme préfixes d'option.
- Prend en charge la répétition de valeurs (zéro ou plus, un ou plus).
- Fournit des messages d'aide plus complets.
- Fournit une interface plus simple pour les types et actions personnalisés

A partial upgrade path from `optparse` to `argparse` :

- Replace all `optparse.OptionParser.add_option()` calls with `ArgumentParser.add_argument()` calls.
- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional `ArgumentParser.add_argument()` calls for the positional arguments. Keep in mind that what was previously called `options`, now in the `argparse` context is called `args`.
- Replace `optparse.OptionParser.disable_interspersed_args()` by setting `nargs` of a positional argument to `argparse.REMAINDER`, or use `parse_known_args()` to collect unparsed argument strings in a separate list.
- Replace callback actions and the `callback_*` keyword arguments with `type` or `action` arguments.
- Replace string names for `type` keyword arguments with the corresponding type objects (e.g. `int`, `float`, `complex`, etc).
- Replace `optparse.Values` with `Namespace` and `optparse.OptionError` and `optparse.OptionValueError` with `ArgumentError`.
- Replace strings with implicit arguments such as `%default` or `%prog` with the standard Python syntax to use dictionaries to format strings, that is, `%(default)s` and `%(prog)s`.
- Replace the `OptionParser` constructor version argument with a call to `parser.add_argument('--version', action='version', version='<the version>')`.

16.5 getopt – Analyseur de style C pour les options de ligne de commande

Code source : [Lib/getopt.py](#)

Note : Le module `getopt` est un analyseur pour les options de ligne de commande dont l'API est conçue pour être familière aux utilisateurs de la fonction C `getopt()`. Les utilisateurs qui ne connaissent pas la fonction `getopt()` ou qui aimeraient écrire moins de code, obtenir une meilleure aide et de meilleurs messages d'erreur devraient utiliser le module `argparse`.

Ce module aide les scripts à analyser les arguments de ligne de commande contenus dans `sys.argv`. Il prend en charge les mêmes conventions que la fonction UNIX `getopt()` (y compris les significations spéciales des arguments de la forme `-` et `--`). De longues options similaires à celles prises en charge par le logiciel GNU peuvent également être utilisées via un troisième argument facultatif.

Ce module fournit deux fonctions et une exception :

`getopt.getopt(args, shortopts, longopts=[])`

Analyse les options de ligne de commande et la liste des paramètres. *args* est la liste d'arguments à analyser, sans la référence principale au programme en cours d'exécution. En général, cela signifie `sys.argv[1:]` (donc que le premier argument contenant le nom du programme n'est pas dans la liste). *shortopts* est la chaîne de lettres d'options que le script doit reconnaître, avec des options qui requièrent un argument suivi d'un signe deux-points (:, donc le même format que la version Unix de `getopt()` utilise).

Note : Contrairement au `getopt()` GNU, après un argument n'appartenant pas à une option, aucun argument ne sera considéré comme appartenant à une option. Ceci est similaire à la façon dont les systèmes UNIX non-GNU fonctionnent.

longopts, si spécifié, doit être une liste de chaînes avec les noms des options longues qui doivent être prises en charge. Le premier `--` ne doit pas figurer dans le nom de l'option. Les options longues qui requièrent un argument doivent être suivies d'un signe égal (=). Les arguments facultatifs ne sont pas pris en charge. Pour accepter uniquement les options longues, *shortopts* doit être une chaîne vide. Les options longues sur la ligne de commande peuvent être reconnues tant qu'elles fournissent un préfixe du nom de l'option qui correspond exactement à l'une des options acceptées. Par exemple, si *longopts* est ['foo', 'frob'], l'option `--fo` correspondra à `--foo`, mais `--f` ne correspondra pas de façon unique, donc `GetoptError` sera levé.

La valeur de retour se compose de deux éléments : le premier est une liste de paires (*option*, *value*), la deuxième est la liste des arguments de programme laissés après que la liste d'options est été dépouillée (il s'agit d'une tranche de fin de *args*). Chaque paire option-valeur retournée a l'option comme premier élément, préfixée avec un trait d'union pour les options courtes (par exemple, `-x`) ou deux tirets pour les options longues (par exemple, `--long-option`), et l'argument option comme deuxième élément, ou une chaîne vide si le option n'a aucun argument. Les options se trouvent dans la liste dans l'ordre dans lequel elles ont été trouvées, permettant ainsi plusieurs occurrences. Les options longues et courtes peuvent être mélangées.

`getopt.gnu_getopt(args, shortopts, longopts=[])`

Cette fonction fonctionne comme `getopt()`, sauf que le mode de *scan* GNU est utilisé par défaut. Cela signifie que les arguments option et non-option peuvent être **intermixés**. La fonction `getopt()` arrête le traitement des options dès qu'un argument de non-option est rencontré.

Si le premier caractère de la chaîne d'options est `+`, ou si la variable d'environnement `POSIIXLY_CORRECT` est définie, le traitement des options s'arrête dès qu'un argument non-option est rencontré.

exception `getopt.GetoptError`

Cette exception est levée lorsqu'une option non reconnue est trouvée dans la liste d'arguments ou lorsqu'une option

nécessitant un argument n'en a pas reçu. L'argument de l'exception est une chaîne de caractères indiquant la cause de l'erreur. Pour les options longues, un argument donné à une option qui n'en exige pas un entraîne également le levage de cette exception. Les attributs `msg` et `opt` donnent le message d'erreur et l'option connexe. S'il n'existe aucune option spécifique à laquelle l'exception se rapporte, `opt` est une chaîne vide.

exception `getopt.error`

Alias pour `GetoptError`; pour la rétrocompatibilité.

Un exemple utilisant uniquement les options de style UNIX :

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

L'utilisation de noms d'options longs est tout aussi simple :

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

Dans un script, l'utilisation typique ressemble à ceci :

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err) # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    # ...
```

(suite sur la page suivante)

(suite de la page précédente)

```
if __name__ == "__main__":
    main()
```

Notez qu'une interface de ligne de commande équivalente peut être produite avec moins de code et des messages d'erreur et d'aide plus informatifs à l'aide du module `argparse` module :

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

Voir aussi :

Module `argparse` Option de ligne de commande alternative et bibliothèque d'analyse d'arguments.

16.6 logging — Fonctionnalités de journalisation pour Python

Code source : [Lib/logging/__init__.py](#)

Important

Cette page contient les informations de référence de l'API. Pour des tutoriels et des discussions sur des sujets plus avancés, voir

- Tutoriel basique
- Tutoriel avancé
- Recettes pour la journalisation

Ce module définit les fonctions et les classes qui mettent en œuvre un système flexible d'enregistrement des événements pour les applications et les bibliothèques.

Le principal avantage de l'API de journalisation fournie par un module de bibliothèque standard est que tous les modules Python peuvent participer à la journalisation, de sorte que le journal de votre application peut inclure vos propres messages intégrés aux messages de modules tiers.

Le module offre beaucoup de fonctionnalités et de flexibilité. Si vous n'êtes pas familiarisé avec la journalisation, la meilleure façon de vous y familiariser est de consulter les tutoriels (voir les liens à droite).

Les classes de base définies par le module, ainsi que leurs fonctions, sont énumérées ci-dessous.

- Les enregistreurs (*loggers* en anglais) exposent l'interface que le code de l'application utilise directement.
- Les gestionnaires (*handlers*) envoient les entrées de journal (créées par les *loggers*) vers les destinations voulues.
- Les filtres (*filters*) fournissent un moyen de choisir finement quelles entrées de journal doivent être sorties.
- Les formateurs (*formatters*) spécifient la structure de l'entrée de journal dans la sortie finale.

16.6.1 Objets Enregistreurs

Loggers have the following attributes and methods. Note that Loggers are never instantiated directly, but always through the module-level function `logging.getLogger(name)`. Multiple calls to `getLogger()` with the same name will always return a reference to the same Logger object.

Le nom `name` est potentiellement une valeur avec plusieurs niveaux de hiérarchie, séparés par des points, comme `truc.machin.bidule` (bien qu'il puisse aussi être simplement `truc`, par exemple). Les enregistreurs qui sont plus bas dans la liste hiérarchique sont les enfants des enregistreurs plus haut dans la liste. Par exemple, pour un enregistreur nommé `truc`, les enregistreurs portant les noms `truc.machin`, `truc.machin.bidule` et `truc.chose` sont tous des descendants de `truc`. La hiérarchie des noms d'enregistreurs est analogue à la hiérarchie des paquets Python, et est identique à celle-ci si vous organisez vos enregistreurs par module en utilisant la construction recommandée `logging.getLogger(__name__)`. C'est ainsi parce que dans un module, `__name__` est le nom du module dans l'espace de noms des paquets Python.

class `logging.Logger`

propagate

Si cet attribut est évalué comme vrai, les événements enregistrés dans cet enregistreur seront transmis aux gestionnaires des enregistreurs de niveau supérieur (parents), en plus des gestionnaires attachés à cet enregistreur. Les messages sont transmis directement aux gestionnaires des enregistreurs parents — ni le niveau ni les filtres des enregistreurs ancestraux en question ne sont pris en compte.

S'il s'évalue comme faux, les messages de journalisation ne sont pas transmis aux gestionnaires des enregistreurs parents.

Le constructeur fixe cet attribut à `True`.

Note : Si vous associez un gestionnaire à un enregistreur *et* à un ou plusieurs de ses parents, il peut émettre le même enregistrement plusieurs fois. En général, vous ne devriez pas avoir besoin d'attacher un gestionnaire à plus d'un enregistreur — si vous l'attachez simplement à l'enregistreur approprié qui est le plus haut dans la hiérarchie des enregistreurs, alors il voit tous les événements enregistrés par tous les enregistreurs descendants, à condition que leur paramètre de propagation soit laissé à `True`. Un scénario courant est d'attacher les gestionnaires uniquement à l'enregistreur racine, et de laisser la propagation s'occuper du reste.

setLevel (*level*)

Fixe le seuil de cet enregistreur au niveau *level*. Les messages de journalisation qui sont moins graves que *level* sont ignorés ; les messages qui ont une gravité égale à *level* ou plus élevée sont émis par le ou les gestionnaires de cet enregistreur, à moins que le niveau d'un gestionnaire n'ait été fixé à un niveau de gravité plus élevé que *level*.

Lorsqu'un enregistreur est créé, le niveau est fixé à `NOTSET` (ce qui entraîne le traitement de tous les messages lorsque l'enregistreur est l'enregistreur racine, ou la délégation au parent lorsque l'enregistreur est un enregistreur non racine). Notez que l'enregistreur racine est créé avec le niveau `WARNING`.

Le terme « délégation au parent » signifie que si un enregistreur a un niveau de `NOTSET`, sa chaîne d'enregistreurs parents est parcourue jusqu'à ce qu'un parent ayant un niveau autre que `NOTSET` soit trouvé, ou que la racine soit atteinte.

Si un ancêtre est trouvé avec un niveau autre que `NOTSET`, alors le niveau de ce parent est traité comme le niveau effectif de l'enregistreur où la recherche de l'ancêtre a commencé, et est utilisé pour déterminer comment un événement d'enregistrement est traité.

Si la racine est atteinte, et qu'elle a un niveau de `NOTSET`, alors tous les messages sont traités. Sinon, le niveau de la racine est utilisé comme niveau effectif.

Voir *Niveaux de journalisation* pour la liste des niveaux.

Modifié dans la version 3.2 : Le paramètre *level* accepte maintenant une représentation du niveau en chaîne de caractères (comme `'INFO'`) en alternative aux constantes entières comme `INFO`. Notez, cependant, que les niveaux sont stockés en interne sous forme d'entiers, et des méthodes telles que `getEffectiveLevel()` et `isEnabledFor()` renvoient/s'attendent à recevoir des entiers.

isEnabledFor (*lvl*)

Indicates if a message of severity *lvl* would be processed by this logger. This method checks first the module-level level set by `logging.disable(lvl)` and then the logger's effective level as determined by `getEffectiveLevel()`.

getEffectiveLevel ()

Indique le niveau effectif pour cet enregistreur. Si une valeur autre que NOTSET a été définie en utilisant `setLevel()`, elle est renvoyée. Sinon, la hiérarchie est parcourue vers la racine jusqu'à ce qu'une valeur autre que NOTSET soit trouvée, et cette valeur est renvoyée. La valeur renvoyée est un entier, généralement l'un de `logging.DEBUG`, `logging.INFO`, etc.

getChild (*suffix*)

Renvoie un enregistreur qui est un enfant de cet enregistreur, tel que déterminé par le suffixe. Ainsi, `logging.getLogger('abc').getChild('def.ghi')` renvoie le même enregistreur que celui renvoyé par `logging.getLogger('abc.def.ghi')`. C'est une méthode de pure commodité, utile lorsque l'enregistreur parent est nommé en utilisant par exemple `__name__` plutôt qu'une chaîne de caractères littérale.

Nouveau dans la version 3.2.

debug (*msg*, **args*, ***kwargs*)

Logs a message with level `DEBUG` on this logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are three keyword arguments in *kwargs* which are inspected : *exc_info*, *stack_info*, and *extra*.

If *exc_info* does not evaluate as false, it causes exception information to be added to the logging message. If an exception tuple (in the format returned by `sys.exc_info()`) or an exception instance is provided, it is used; otherwise, `sys.exc_info()` is called to get the exception information.

The second optional keyword argument is *stack_info*, which defaults to `False`. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying *exc_info* : The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

You can specify *stack_info* independently of *exc_info*, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says :

```
Stack (most recent call last):
```

This mimics the `Traceback (most recent call last) :` which is used when displaying exception frames.

The third keyword argument is *extra* which can be used to pass a dictionary which is used to populate the `__dict__` of the `LogRecord` created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example :

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: connection_
↪reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the `Formatter` documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the `Formatter` has been set up with a format string which expects “clientip” and “user” in the

attribute dictionary of the `LogRecord`. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the *extra* dictionary with these keys. While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized *Formatters* would be used with particular *Handlers*.

Nouveau dans la version 3.2 : The *stack_info* parameter was added.

Modifié dans la version 3.5 : The *exc_info* parameter can now accept exception instances.

info (*msg, *args, **kwargs*)

Logs a message with level INFO on this logger. The arguments are interpreted as for *debug()*.

warning (*msg, *args, **kwargs*)

Logs a message with level WARNING on this logger. The arguments are interpreted as for *debug()*.

Note : There is an obsolete method `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

error (*msg, *args, **kwargs*)

Logs a message with level ERROR on this logger. The arguments are interpreted as for *debug()*.

critical (*msg, *args, **kwargs*)

Logs a message with level CRITICAL on this logger. The arguments are interpreted as for *debug()*.

log (*lvl, msg, *args, **kwargs*)

Logs a message with integer level *lvl* on this logger. The other arguments are interpreted as for *debug()*.

exception (*msg, *args, **kwargs*)

Logs a message with level ERROR on this logger. The arguments are interpreted as for *debug()*. Exception info is added to the logging message. This method should only be called from an exception handler.

addFilter (*filter*)

Adds the specified filter *filter* to this logger.

removeFilter (*filter*)

Removes the specified filter *filter* from this logger.

filter (*record*)

Applies this logger's filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

addHandler (*hdlr*)

Adds the specified handler *hdlr* to this logger.

removeHandler (*hdlr*)

Removes the specified handler *hdlr* from this logger.

findCaller (*stack_info=False*)

Finds the caller's source filename and line number. Returns the filename, line number, function name and stack information as a 4-element tuple. The stack information is returned as `None` unless *stack_info* is `True`.

handle (*record*)

Handles a record by passing it to all handlers associated with this logger and its ancestors (until a false value of *propagate* is found). This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied using *filter()*.

makeRecord (*name, lvl, fn, lno, msg, args, exc_info, func=None, extra=None, sinfo=None*)

This is a factory method which can be overridden in subclasses to create specialized *LogRecord* instances.

hasHandlers ()

Checks to see if this logger has any handlers configured. This is done by looking for handlers in this logger and its parents in the logger hierarchy. Returns `True` if a handler was found, else `False`. The method stops searching up the hierarchy whenever a logger with the "propagate" attribute set to false is found - that will be the last logger which is checked for the existence of handlers.

Nouveau dans la version 3.2.

Modifié dans la version 3.7 : Loggers can now be pickled and unpickled.

16.6.2 Niveaux de journalisation

Les valeurs numériques des niveaux de journalisation sont données dans le tableau suivant. Celles-ci n'ont d'intérêt que si vous voulez définir vos propres niveaux, avec des valeurs spécifiques par rapport aux niveaux prédéfinis. Si vous définissez un niveau avec la même valeur numérique, il écrase la valeur prédéfinie ; le nom prédéfini est perdu.

Niveau	Valeur numérique
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

16.6.3 Handler Objects

Handlers have the following attributes and methods. Note that *Handler* is never instantiated directly ; this class acts as a base for more useful subclasses. However, the `__init__()` method in subclasses needs to call *Handler*.
`__init__()`.

class `logging.Handler`

__init__ (*level*=*NOTSET*)

Initializes the *Handler* instance by setting its level, setting the list of filters to the empty list and creating a lock (using *createLock()*) for serializing access to an I/O mechanism.

createLock ()

Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

acquire ()

Acquires the thread lock created with *createLock()*.

release ()

Releases the thread lock acquired with *acquire()*.

setLevel (*level*)

Sets the threshold for this handler to *level*. Logging messages which are less severe than *level* will be ignored. When a handler is created, the level is set to NOTSET (which causes all messages to be processed).

Voir *Niveaux de journalisation* pour la liste des niveaux.

Modifié dans la version 3.2 : The *level* parameter now accepts a string representation of the level such as "INFO" as an alternative to the integer constants such as INFO.

setFormatter (*fmt*)

Sets the *Formatter* for this handler to *fmt*.

addFilter (*filter*)

Adds the specified filter *filter* to this handler.

removeFilter (*filter*)

Removes the specified filter *filter* from this handler.

filter (*record*)

Applies this handler's filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

flush()

Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

close()

Tidy up any resources used by the handler. This version does no output but removes the handler from an internal list of handlers which is closed when `shutdown()` is called. Subclasses should ensure that this gets called from overridden `close()` methods.

handle(record)

Conditionally emits the specified logging record, depending on filters which may have been added to the handler. Wraps the actual emission of the record with acquisition/release of the I/O thread lock.

handleError(record)

This method should be called from handlers when an exception is encountered during an `emit()` call. If the module-level attribute `raiseExceptions` is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The specified record is the one which was being processed when the exception occurred. (The default value of `raiseExceptions` is `True`, as that is more useful during development).

format(record)

Do formatting for a record - if a formatter is set, use it. Otherwise, use the default formatter for the module.

emit(record)

Do whatever it takes to actually log the specified logging record. This version is intended to be implemented by subclasses and so raises a `NotImplementedError`.

For a list of handlers included as standard, see `logging.handlers`.

16.6.4 Formatter Objects

`Formatter` objects have the following attributes and methods. They are responsible for converting a `LogRecord` to (usually) a string which can be interpreted by either a human or an external system. The base `Formatter` allows a formatting string to be specified. If none is supplied, the default value of `'%(message)s'` is used, which just includes the message in the logging call. To have additional items of information in the formatted output (such as a timestamp), keep reading.

A Formatter can be initialized with a format string which makes use of knowledge of the `LogRecord` attributes - such as the default value mentioned above making use of the fact that the user's message and arguments are pre-formatted into a `LogRecord`'s `message` attribute. This format string contains standard Python %-style mapping keys. See section *Formatage de chaînes à la printf* for more information on string formatting.

The useful mapping keys in a `LogRecord` are given in the section on *LogRecord attributes*.

class logging.Formatter (*fmt=None, datefmt=None, style='%*)

Returns a new instance of the `Formatter` class. The instance is initialized with a format string for the message as a whole, as well as a format string for the date/time portion of a message. If no *fmt* is specified, `'%(message)s'` is used. If no *datefmt* is specified, a format is used which is described in the `formatTime()` documentation.

The *style* parameter can be one of `"%"`, `"{"` or `"$"` and determines how the format string will be merged with its data : using one of %-formatting, `str.format()` or `string.Template`. See formatting-styles for more information on using `{-` and `$-` formatting for log messages.

Modifié dans la version 3.2 : The *style* parameter was added.

format(record)

The record's attribute dictionary is used as the operand to a string formatting operation. Returns the resulting string. Before formatting the dictionary, a couple of preparatory steps are carried out. The `message` attribute of the record is computed using `msg % args`. If the formatting string contains `'(asctime)'`, `formatTime()` is called to format the event time. If there is exception information, it is formatted using `formatException()` and appended to the message. Note that the formatted exception information is

cached in attribute `exc_text`. This is useful because the exception information can be pickled and sent across the wire, but you should be careful if you have more than one `Formatter` subclass which customizes the formatting of exception information. In this case, you will have to clear the cached value after a formatter has done its formatting, so that the next formatter to handle the event doesn't use the cached value but recalculates it afresh.

If stack information is available, it's appended after the exception information, using `formatStack()` to transform it if necessary.

formatTime (*record*, *datefmt=None*)

This method should be called from `format()` by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior is as follows : if *datefmt* (a string) is specified, it is used with `time.strftime()` to format the creation time of the record. Otherwise, the format “%Y-%m-%d %H:%M:%S,uuu” is used, where the uuu part is a millisecond value and the other letters are as per the `time.strftime()` documentation. An example time in this format is 2003-01-23 00:29:50,411. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, `time.localtime()` is used; to change this for a particular formatter instance, set the `converter` attribute to a function with the same signature as `time.localtime()` or `time.gmtime()`. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the `Formatter` class.

Modifié dans la version 3.3 : Previously, the default format was hard-coded as in this example : 2010-09-06 22:38:15,292 where the part before the comma is handled by a strftime format string ('%Y-%m-%d %H:%M:%S'), and the part after the comma is a millisecond value. Because strftime does not have a format placeholder for milliseconds, the millisecond value is appended using another format string, '%s,%03d' — and both of these format strings have been hardcoded into this method. With the change, these strings are defined as class-level attributes which can be overridden at the instance level when desired. The names of the attributes are `default_time_format` (for the strftime format string) and `default_msec_format` (for appending the millisecond value).

formatException (*exc_info*)

Formats the specified exception information (a standard exception tuple as returned by `sys.exc_info()`) as a string. This default implementation just uses `traceback.print_exception()`. The resulting string is returned.

formatStack (*stack_info*)

Formats the specified stack information (a string as returned by `traceback.print_stack()`, but with the last newline removed) as a string. This default implementation just returns the input value.

16.6.5 Filter Objects

Filters can be used by Handlers and Loggers for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with “A.B” will allow events logged by loggers “A.B”, “A.B.C”, “A.B.C.D”, “A.B.D” etc. but not “A.BB”, “B.A.B” etc. If initialized with the empty string, all events are passed.

class `logging.Filter` (*name=""*)

Returns an instance of the `Filter` class. If *name* is specified, it names a logger which, together with its children, will have its events allowed through the filter. If *name* is the empty string, allows every event.

filter (*record*)

Is the specified record to be logged ? Returns zero for no, nonzero for yes. If deemed appropriate, the record may be modified in-place by this method.

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using `debug()`, `info()`, etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger's filter setting, unless the filter has also been applied to those descendant loggers.

You don't actually need to subclass `Filter` : you can pass any instance which has a `filter` method with the same semantics.

Modifié dans la version 3.2 : You don't need to create specialized `Filter` classes, or use other classes with a `filter` method : you can use a function (or other callable) as a filter. The filtering logic will check to see if the filter object has a `filter` attribute : if it does, it's assumed to be a `Filter` and its `filter()` method is called. Otherwise, it's assumed to be a callable and called with the record as the single parameter. The returned value should conform to that returned by `filter()`.

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they're attached to : this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the `LogRecord` being processed. Obviously changing the `LogRecord` needs to be done with some care, but it does allow the injection of contextual information into logs (see `filters-contextual`).

16.6.6 LogRecord Objects

`LogRecord` instances are created automatically by the `Logger` every time something is logged, and can be created manually via `makeLogRecord()` (for example, from a pickled event received over the wire).

class `logging.LogRecord` (*name, level, pathname, lineno, msg, args, exc_info, func=None, sinfo=None*)

Contains all the information pertinent to the event being logged.

The primary information is passed in `msg` and `args`, which are combined using `msg % args` to create the message field of the record.

Paramètres

- **name** – The name of the logger used to log the event represented by this `LogRecord`. Note that this name will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.
- **level** – The numeric level of the logging event (one of `DEBUG`, `INFO` etc.) Note that this is converted to *two* attributes of the `LogRecord` : `levelno` for the numeric value and `levelname` for the corresponding level name.
- **pathname** – The full pathname of the source file where the logging call was made.
- **lineno** – The line number in the source file where the logging call was made.
- **msg** – The event description message, possibly a format string with placeholders for variable data.
- **args** – Variable data to merge into the `msg` argument to obtain the event description.
- **exc_info** – An exception tuple with the current exception information, or `None` if no exception information is available.
- **func** – The name of the function or method from which the logging call was invoked.
- **sinfo** – A text string representing stack information from the base of the stack in the current thread, up to the logging call.

`getMessage()`

Returns the message for this `LogRecord` instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, `str()` is called on it to convert it to a string. This allows use of user-defined classes as messages, whose `__str__` method can return the actual format string to be used.

Modifié dans la version 3.2 : The creation of a `LogRecord` has been made more configurable by providing a factory which is used to create the record. The factory can be set using `getLogRecordFactory()` and `setLogRecordFactory()` (see this for the factory's signature).

This functionality can be used to inject your own values into a `LogRecord` at creation time. You can use the following pattern :

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

With this pattern, multiple factories could be chained, and as long as they don't overwrite each other's attributes or unintentionally overwrite the standard attributes listed above, there should be no surprises.

16.6.7 LogRecord attributes

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

If you are using {}-formatting (*str.format()*), you can use {attrname} as the placeholder in the format string. If you are using \$-formatting (*string.Template*), use the form \${attrname}. In both cases, of course, replace attrname with the actual attribute name you want to use.

In the case of {}-formatting, you can specify formatting flags by placing them after the attribute name, separated from it with a colon. For example : a placeholder of {msecs:03d} would format a millisecond value of 4 as 004. Refer to the *str.format()* documentation for full details on the options available to you.

Attribute name	Format	Description
args	You shouldn't need to format this yourself.	The tuple of arguments merged into <code>msg</code> to produce <code>message</code> , or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	<code>%(asctime)s</code>	Human-readable time when the <i>LogRecord</i> was created. By default this is of the form "2003-07-08 16 :49 :45,896" (the numbers after the comma are millisecond portion of the time).
created	<code>%(created)f</code>	Time when the <i>LogRecord</i> was created (as returned by <code>time.time()</code>).
exc_info	You shouldn't need to format this yourself.	Exception tuple (à la <code>sys.exc_info</code>) or, if no exception has occurred, <code>None</code> .
filename	<code>%(filename)s</code>	Filename portion of <code>pathname</code> .
funcName	<code>%(funcName)s</code>	Name of function containing the logging call.
levelname	<code>%(levelname)s</code>	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').
levelno	<code>%(levelno)s</code>	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).
lineno	<code>%(lineno)d</code>	Source line number where the logging call was issued (if available).
message	<code>%(message)s</code>	The logged message, computed as <code>msg % args</code> . This is set when <i>Formatter.format()</i> is invoked.
module	<code>%(module)s</code>	Module (name portion of <code>filename</code>).
msecs	<code>%(msecs)d</code>	Millisecond portion of the time when the <i>LogRecord</i> was created.
msg	You shouldn't need to format this yourself.	The format string passed in the original logging call. Merged with <code>args</code> to produce <code>message</code> , or an arbitrary object (see arbitrary-object-messages).
name	<code>%(name)s</code>	Name of the logger used to log the call.
pathname	<code>%(pathname)s</code>	Full pathname of the source file where the logging call was issued (if available).
process	<code>%(process)d</code>	Process ID (if available).
processName	<code>%(processName)s</code>	Process name (if available).
relativeCreated	<code>%(relativeCreated)d</code>	Time in milliseconds when the <i>LogRecord</i> was created, relative to the time the logging module was loaded.
stack_info	You shouldn't need to format this yourself.	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	<code>%(thread)d</code>	Thread ID (if available).
threadName	<code>%(threadName)s</code>	Thread name (if available).

Modifié dans la version 3.1 : *processName* was added.

16.6.8 LoggerAdapter Objects

LoggerAdapter instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on adding contextual information to your logging output.

class `logging.LoggerAdapter` (*logger*, *extra*)

Returns an instance of *LoggerAdapter* initialized with an underlying *Logger* instance and a dict-like object.

process (*msg*, *kwargs*)

Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual information. This implementation takes the object passed as *extra* to the constructor and adds it to *kwargs* using key “extra”. The return value is a (*msg*, *kwargs*) tuple which has the (possibly modified) versions of the arguments passed in.

In addition to the above, *LoggerAdapter* supports the following methods of *Logger* : *debug()*, *info()*, *warning()*, *error()*, *exception()*, *critical()*, *log()*, *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *hasHandlers()*. These methods have the same signatures as their counterparts in *Logger*, so you can use the two types of instances interchangeably.

Modifié dans la version 3.2 : The *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *hasHandlers()* methods were added to *LoggerAdapter*. These methods delegate to the underlying logger.

16.6.9 Thread Safety

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this though using threading locks; there is one lock to serialize access to the module’s shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the *signal* module, you may not be able to use logging from within such handlers. This is because lock implementations in the *threading* module are not always re-entrant, and so cannot be invoked from such signal handlers.

16.6.10 Fonctions de niveau module

In addition to the classes described above, there are a number of module-level functions.

`logging.getLogger` (*name=None*)

Return a logger with the specified name or, if *name* is *None*, return a logger which is the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like “a”, “a.b” or “a.b.c.d”. Choice of these names is entirely up to the developer who is using logging.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

`logging.getLoggerClass` ()

Return either the standard *Logger* class, or the last class passed to *setLoggerClass()*. This function may be called from within a new class definition, to ensure that installing a customized *Logger* class will not undo customizations already applied by other code. For example :

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

`logging.getLogRecordFactory` ()

Return a callable which is used to create a *LogRecord*.

Nouveau dans la version 3.2 : This function has been provided, along with *setLogRecordFactory()*, to allow developers more control over how the *LogRecord* representing a logging event is constructed.

See *setLogRecordFactory()* for more information about the how the factory is called.

`logging.debug(msg, *args, **kwargs)`

Logs a message with level `DEBUG` on the root logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are three keyword arguments in *kwargs* which are inspected : *exc_info* which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by `sys.exc_info()`) or an exception instance is provided, it is used ; otherwise, `sys.exc_info()` is called to get the exception information.

The second optional keyword argument is *stack_info*, which defaults to `False`. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying *exc_info* : The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

You can specify *stack_info* independently of *exc_info*, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says :

```
Stack (most recent call last):
```

This mimics the `Traceback (most recent call last):` which is used when displaying exception frames.

The third optional keyword argument is *extra* which can be used to pass a dictionary which is used to populate the `__dict__` of the `LogRecord` created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example :

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning('Protocol problem: %s', 'connection reset', extra=d)
```

would print something like :

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: connection reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the *Formatter* documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the *Formatter* has been set up with a format string which expects “clientip” and “user” in the attribute dictionary of the `LogRecord`. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the *extra* dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized *Formatters* would be used with particular *Handlers*.

Nouveau dans la version 3.2 : The *stack_info* parameter was added.

`logging.info(msg, *args, **kwargs)`

Logs a message with level `INFO` on the root logger. The arguments are interpreted as for `debug()`.

`logging.warning(msg, *args, **kwargs)`

Logs a message with level `WARNING` on the root logger. The arguments are interpreted as for `debug()`.

Note : There is an obsolete function `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

`logging.error(msg, *args, **kwargs)`

Logs a message with level `ERROR` on the root logger. The arguments are interpreted as for `debug()`.

`logging.critical(msg, *args, **kwargs)`

Logs a message with level `CRITICAL` on the root logger. The arguments are interpreted as for `debug()`.

`logging.exception(msg, *args, **kwargs)`

Logs a message with level `ERROR` on the root logger. The arguments are interpreted as for `debug()`. Exception info is added to the logging message. This function should only be called from an exception handler.

`logging.log(level, msg, *args, **kwargs)`

Logs a message with level `level` on the root logger. The other arguments are interpreted as for `debug()`.

Note : The above module-level convenience functions, which delegate to the root logger, call `basicConfig()` to ensure that at least one handler is available. Because of this, they should *not* be used in threads, in versions of Python earlier than 2.7.1 and 3.2, unless at least one handler has been added to the root logger *before* the threads are started. In earlier versions of Python, due to a thread safety shortcoming in `basicConfig()`, this can (under rare circumstances) lead to handlers being added multiple times to the root logger, which can in turn lead to multiple messages for the same event.

`logging.disable(lvl=CRITICAL)`

Provides an overriding level `lvl` for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity `lvl` and below, so that if you call it with a value of `INFO`, then all `INFO` and `DEBUG` events would be discarded, whereas those of severity `WARNING` and above would be processed according to the logger's effective level. If `logging.disable(logging.NOTSET)` is called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

Note that if you have defined any custom logging level higher than `CRITICAL` (this is not recommended), you won't be able to rely on the default value for the `lvl` parameter, but will have to explicitly supply a suitable value.

Modifié dans la version 3.7 : The `lvl` parameter was defaulted to level `CRITICAL`. See Issue #28524 for more information about this change.

`logging.addLevelName(lvl, levelName)`

Associates level `lvl` with text `levelName` in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a `Formatter` formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

Note : If you are thinking of defining your own levels, please see the section on custom-levels.

`logging.getLevelName(lvl)`

Returns the textual representation of logging level `lvl`. If the level is one of the predefined levels `CRITICAL`, `ERROR`, `WARNING`, `INFO` or `DEBUG` then you get the corresponding string. If you have associated levels with names using `addLevelName()` then the name you have associated with `lvl` is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned. Otherwise, the string "Level %s" % `lvl` is returned.

Note : Levels are internally integers (as they need to be compared in the logging logic). This function is used to convert between an integer level and the level name displayed in the formatted log output by means of the `%(levelname)s` format specifier (see *LogRecord attributes*).

Modifié dans la version 3.4 : In Python versions earlier than 3.4, this function could also be passed a text level, and would return the corresponding numeric value of the level. This undocumented behaviour was considered a mistake, and was removed in Python 3.4, but reinstated in 3.4.2 due to retain backward compatibility.

`logging.makeLogRecord(attdict)`

Creates and returns a new *LogRecord* instance whose attributes are defined by `attdict`. This function is useful

for taking a pickled *LogRecord* attribute dictionary, sent over a socket, and reconstituting it as a *LogRecord* instance at the receiving end.

```
logging.basicConfig(**kwargs)
```

Does basic configuration for the logging system by creating a *StreamHandler* with a default *Formatter* and adding it to the root logger. The functions *debug()*, *info()*, *warning()*, *error()* and *critical()* will call *basicConfig()* automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured for it.

Note : This function should be called from the main thread before other threads are started. In versions of Python prior to 2.7.1 and 3.2, if this function is called from multiple threads, it is possible (in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

The following keyword arguments are supported.

Format	Description
<i>filename</i>	Specifies that a FileHandler be created, using the specified filename, rather than a StreamHandler.
<i>filemode</i>	If <i>filename</i> is specified, open the file in this <i>mode</i> . Defaults to 'a'.
<i>format</i>	Use the specified format string for the handler.
<i>datefmt</i>	Use the specified date/time format, as accepted by <i>time.strftime()</i> .
<i>style</i>	If <i>format</i> is specified, use this style for the format string. One of '%', '{' or '\$' for <i>printf-style</i> , <i>str.format()</i> or <i>string.Template</i> respectively. Defaults to '% '.
<i>level</i>	Set the root logger level to the specified <i>level</i> .
<i>stream</i>	Use the specified stream to initialize the StreamHandler. Note that this argument is incompatible with <i>filename</i> - if both are present, a <code>ValueError</code> is raised.
<i>handlers</i>	If specified, this should be an iterable of already created handlers to add to the root logger. Any handlers which don't already have a formatter set will be assigned the default formatter created in this function. Note that this argument is incompatible with <i>filename</i> or <i>stream</i> - if both are present, a <code>ValueError</code> is raised.

Modifié dans la version 3.2 : The *style* argument was added.

Modifié dans la version 3.3 : The *handlers* argument was added. Additional checks were added to catch situations where incompatible arguments are specified (e.g. *handlers* together with *stream* or *filename*, or *stream* together with *filename*).

```
logging.shutdown()
```

Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should be called at application exit and no further use of the logging system should be made after this call.

```
logging.setLoggerClass(klass)
```

Tells the logging system to use the class *klass* when instantiating a logger. The class should define `__init__()` such that only a name argument is required, and the `__init__()` should call `Logger.__init__()`. This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior.

```
logging.setLogRecordFactory(factory)
```

Set a callable which is used to create a *LogRecord*.

Paramètres **factory** – The factory callable to be used to instantiate a log record.

Nouveau dans la version 3.2 : This function has been provided, along with `getLogRecordFactory()`, to allow developers more control over how the `LogRecord` representing a logging event is constructed.

The factory has the following signature :

```
factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None,
**kwargs)
```

name The logger name.

level The logging level (numeric).
fn The full pathname of the file where the logging call was made.
lno The line number in the file where the logging call was made.
msg The logging message.
args The arguments for the logging message.
exc_info An exception tuple, or `None`.
func The name of the function or method which invoked the logging call.
sinfo A stack traceback such as is provided by `traceback.print_stack()`, showing the call hierarchy.
kwargs Additional keyword arguments.

16.6.11 Module-Level Attributes

`logging.lastResort`

A « handler of last resort » is available through this attribute. This is a *StreamHandler* writing to `sys.stderr` with a level of `WARNING`, and is used to handle logging events in the absence of any logging configuration. The end result is to just print the message to `sys.stderr`. This replaces the earlier error message saying that « no handlers could be found for logger XYZ ». If you need the earlier behaviour for some reason, `lastResort` can be set to `None`.

Nouveau dans la version 3.2.

16.6.12 Integration with the warnings module

The `captureWarnings()` function can be used to integrate *logging* with the *warnings* module.

`logging.captureWarnings(capture)`

This function is used to turn the capture of warnings by logging on and off.

If `capture` is `True`, warnings issued by the *warnings* module will be redirected to the logging system. Specifically, a warning will be formatted using `warnings.formatwarning()` and the resulting string logged to a logger named `'py.warnings'` with a severity of `WARNING`.

If `capture` is `False`, the redirection of warnings to the logging system will stop, and warnings will be redirected to their original destinations (i.e. those in effect before `captureWarnings(True)` was called).

Voir aussi :

Module *logging.config* API de configuration pour le module de journalisation.

Module *logging.handlers* Gestionnaires utiles inclus avec le module de journalisation.

PEP 282 - A Logging System The proposal which described this feature for inclusion in the Python standard library.

Original Python logging package This is the original source for the *logging* package. The version of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the *logging* package in the standard library.

16.7 logging.config — Logging configuration

Source code : [Lib/logging/config.py](#)

Important

This page contains only reference information. For tutorials, please see

- [Basic Tutorial](#)
- [Advanced Tutorial](#)
- [Logging Cookbook](#)

This section describes the API for configuring the logging module.

16.7.1 Configuration functions

The following functions configure the logging module. They are located in the `logging.config` module. Their use is optional — you can configure the logging module using these functions or by making calls to the main API (defined in `logging` itself) and defining handlers which are declared either in `logging` or `logging.handlers`.

`logging.config.dictConfig(config)`

Takes the logging configuration from a dictionary. The contents of this dictionary are described in [Configuration dictionary schema](#) below.

If an error is encountered during configuration, this function will raise a `ValueError`, `TypeError`, `AttributeError` or `ImportError` with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error :

- A `level` which is not a string or which is a string not corresponding to an actual logging level.
- A `propagate` value which is not a boolean.
- An `id` which does not have a corresponding destination.
- A non-existent handler `id` found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the `DictConfigurator` class, whose constructor is passed the dictionary used for configuration, and has a `configure()` method. The `logging.config` module has a callable attribute `dictConfigClass` which is initially set to `DictConfigurator`. You can replace the value of `dictConfigClass` with a suitable implementation of your own.

`dictConfig()` calls `dictConfigClass` passing the specified dictionary, and then calls the `configure()` method on the returned object to put the configuration into effect :

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of `DictConfigurator` could call `DictConfigurator.__init__()` in its own `__init__()`, then set up custom prefixes which would be usable in the subsequent `configure()` call. `dictConfigClass` would be bound to this new subclass, and then `dictConfig()` could be called exactly as in the default, uncustomized state.

Nouveau dans la version 3.2.

`logging.config.fileConfig(fname, defaults=None, disable_existing_loggers=True)`

Reads the logging configuration from a `configparser`-format file. The format of the file should be as described in [Configuration file format](#). This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

Paramètres

- **fname** – A filename, or a file-like object, or an instance derived from `RawConfigParser`. If a `RawConfigParser`-derived instance is passed, it is used as is. Otherwise, a `ConfigParser` is instantiated, and the configuration read by it from the object passed in `fname`. If that has a `readline()` method, it is assumed to be a file-like object and read using `read_file()`; otherwise, it is assumed to be a filename and passed to `read()`.
- **defaults** – Defaults to be passed to the `ConfigParser` can be specified in this argument.
- **disable_existing_loggers** – If specified as `False`, loggers which exist when this call is made are left enabled. The default is `True` because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing loggers unless they or their ancestors are explicitly named in the logging configuration.

Modifié dans la version 3.4 : An instance of a subclass of `RawConfigParser` is now accepted as a value for `fname`. This facilitates :

- Use of a configuration file where logging configuration is just part of the overall application configuration.
- Use of a configuration read from a file, and then modified by the using application (e.g. based on command-line parameters or other aspects of the runtime environment) before being passed to `fileConfig`.

`logging.config.listen(port=DEFAULT_LOGGING_CONFIG_PORT, verify=None)`

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module's default `DEFAULT_LOGGING_CONFIG_PORT` is used. Logging configurations will be sent as a file suitable for processing by `dictConfig()` or `fileConfig()`. Returns a `Thread` instance on which you can call `start()` to start the server, and which you can `join()` when appropriate. To stop the server, call `stopListening()`.

The `verify` argument, if specified, should be a callable which should verify whether bytes received across the socket are valid and should be processed. This could be done by encrypting and/or signing what is sent across the socket, such that the `verify` callable can perform signature verification and/or decryption. The `verify` callable is called with a single argument - the bytes received across the socket - and should return the bytes to be processed, or `None` to indicate that the bytes should be discarded. The returned bytes could be the same as the passed in bytes (e.g. when only verification is done), or they could be completely different (perhaps if decryption were performed). To send a configuration to the socket, read in the configuration file and send it to the socket as a sequence of bytes preceded by a four-byte length string packed in binary using `struct.pack('>L', n)`.

Note : Because portions of the configuration are passed through `eval()`, use of this function may open its users to a security risk. While the function only binds to a socket on `localhost`, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls `listen()`. Specifically, if the process calling `listen()` runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's `listen()` socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used). To avoid the risk of this happening, use the `verify` argument to `listen()` to prevent unrecognised configurations from being applied.

Modifié dans la version 3.4 : The `verify` argument was added.

Note : If you want to send configurations to the listener which don't disable existing loggers, you will need to use a JSON format for the configuration, which will use `dictConfig()` for configuration. This method allows you to specify `disable_existing_loggers` as `False` in the configuration you send.

`logging.config.stopListening()`

Stops the listening server which was created with a call to `listen()`. This is typically called before calling `join()` on the return value from `listen()`.

16.7.2 Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named “console” and then say that the logger named “startup” will send its messages to the “console” handler. These objects aren’t limited to those provided by the `logging` module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as `sys.stderr`. The syntax for describing these objects and connections is defined in *Object connections* below.

Dictionary Schema Details

The dictionary passed to `dictConfig()` must contain the following keys :

- `version` - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a “configuring dict” is mentioned, it will be checked for the special `'()'` key to see if a custom instantiation is required. If so, the mechanism described in *User-defined objects* below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- `formatters` - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding `Formatter` instance.
The configuring dict is searched for keys `format` and `datefmt` (with defaults of `None`) and these are used to construct a `Formatter` instance.
- `filters` - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.
The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a `logging.Filter` instance.
- `handlers` - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.
The configuring dict is searched for the following keys :
 - `class` (mandatory). This is the fully qualified name of the handler class.
 - `level` (optional). The level of the handler.
 - `formatter` (optional). The id of the formatter for this handler.
 - `filters` (optional). A list of ids of the filters for this handler.

All *other* keys are passed through as keyword arguments to the handler’s constructor. For example, given the snippet :

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level  : INFO
    filters: [allow_foo]
    stream : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

the handler with id `console` is instantiated as a `logging.StreamHandler`, using `sys.stdout` as the underlying stream. The handler with id `file` is instantiated as a `logging.handlers.RotatingFileHandler` with the keyword arguments `filename='logconfig.log'`, `maxBytes=1024`, `backupCount=3`.

- `loggers` - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

The configuring dict is searched for the following keys :

- `level` (optional). The level of the logger.
- `propagate` (optional). The propagation setting of the logger.
- `filters` (optional). A list of ids of the filters for this logger.
- `handlers` (optional). A list of ids of the handlers for this logger.

The specified loggers will be configured according to the level, propagation, filters and handlers specified.

- `root` - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the `propagate` setting will not be applicable.
- `incremental` - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to `False`, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing `fileConfig()` API.

If the specified value is `True`, the configuration is processed as described in the section on [Incremental Configuration](#).

- `disable_existing_loggers` - whether any existing loggers are to be disabled. This setting mirrors the parameter of the same name in `fileConfig()`. If absent, this parameter defaults to `True`. This value is ignored if `incremental` is `True`.

Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the `incremental` key of a configuration dict is present and is `True`, the system will completely ignore any `formatters` and `filters` entries, and process only the `level` settings in the `handlers` entries, and the `level` and `propagate` settings in the `loggers` and `root` entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.

Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object's configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet :

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
```

(suite sur la page suivante)

(suite de la page précédente)

```

h1: #This is an id
    # configuration of handler with id 'h1' goes here
    formatter: brief
h2: #This is another id
    # configuration of handler with id 'h2' goes here
    formatter: precise
loggers:
    foo.bar.baz:
        # other configuration for logger 'foo.bar.baz'
        handlers: [h1, h2]

```

(Note : YAML used here because it's a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. `foo.bar.baz`. The ids for Formatters and Filters can be any string value (such as `brief`, `precise` above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named `foo.bar.baz` should have two handlers attached to it, which are described by the handler ids `h1` and `h2`. The formatter for `h1` is that described by id `brief`, and the formatter for `h2` is that described by id `precise`.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a “factory” - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key `'()'`. Here's a concrete example :

```

formatters:
    brief:
        format: '%(message)s'
    default:
        format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
        datefmt: '%Y-%m-%d %H:%M:%S'
    custom:
        (): my.package.customFormatterFactory
        bar: baz
        spam: 99.9
        answer: 42

```

The above YAML snippet defines three formatters. The first, with id `brief`, is a standard `logging.Formatter` instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a `logging.Formatter` initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries :

```

{
    'format' : '%(message)s'
}

```

et :


```
{
    'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
    'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectively, and as these dictionaries do not contain the special key '()' , the instantiation is inferred from the context : as a result, standard `logging.Formatter` instances are created. The configuration sub-dictionary for the third formatter, with id `custom`, is :

```
{
    '()' : 'my.package.customFormatterFactory',
    'bar' : 'baz',
    'spam' : 99.9,
    'answer' : 42
}
```

and this contains the special key '()' , which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the **remaining** items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id `custom` will be assumed to be returned by the call :

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

The key '()' has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The '()' also serves as a mnemonic that the corresponding value is a callable.

Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string `'sys.stderr'`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `'ext://sys.stderr'` is provided as a value in the configuration, then the `ext://` will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling : there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$` whereby, if the `prefix` is recognised, the `suffix` is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value `'DEBUG'` for a level in a logger or handler will automatically be converted to the value `logging.DEBUG`, and the `handlers`, `filters` and `formatter` entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the `logging` module. For example, consider `logging.handlers.MemoryHandler`, which takes a `target` argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given `target` just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however,

a user defines a `my.package.MyHandler` which has an alternate handler, the configuration system would not know that the alternate referred to a handler. To cater for this, a generic resolution system allows the user to specify :

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

The literal string `'cfg://handlers.file'` will be resolved in an analogous way to strings with the `ext://` prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by `str.format`. Thus, given the following snippet :

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

in the configuration, the string `'cfg://handlers'` would resolve to the dict with key `handlers`, the string `'cfg://handlers.email'` would resolve to the dict with key `email` in the `handlers` dict, and so on. The string `'cfg://handlers.email.toaddrs[1]'` would resolve to `'dev_team.domain.tld'` and the string `'cfg://handlers.email.toaddrs[0]'` would resolve to the value `'support_team@domain.tld'`. The subject value could be accessed using either `'cfg://handlers.email.subject'` or, equivalently, `'cfg://handlers.email[subject]'`. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string `cfg://handlers.myhandler.mykey.123`, this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the string is specified as `cfg://handlers.myhandler.mykey[123]`, the system will attempt to retrieve the value from `config_dict['handlers']['myhandler']['mykey'][123]`, and fall back to `config_dict['handlers']['myhandler']['mykey']['123']` if that fails.

Import resolution and custom importers

Import resolution, by default, uses the builtin `__import__()` function to do its importing. You may want to replace this with your own importing mechanism : if so, you can replace the `importer` attribute of the `DictConfigurator` or its superclass, the `BaseConfigurator` class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with `staticmethod()`. For example :

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

You don't need to wrap with `staticmethod()` if you're setting the import callable on a configurator *instance*.

16.7.3 Configuration file format

The configuration file format understood by `fileConfig()` is based on `configparser` functionality. The file must contain sections called `[loggers]`, `[handlers]` and `[formatters]` which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named `log01` in the `[loggers]` section, the relevant configuration details are held in a section `[logger_log01]`. Similarly, a handler called `hand01` in the `[handlers]` section will have its configuration held in a section called `[handler_hand01]`, while a formatter called `form01` in the `[formatters]` section will have its configuration specified in a section called `[formatter_form01]`. The root logger configuration must be specified in a section called `[logger_root]`.

Note : The `fileConfig()` API is older than the `dictConfig()` API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure `Filter` objects, which provide for filtering of messages beyond simple integer levels, using `fileConfig()`. If you need to have instances of `Filter` in your logging configuration, you will need to use `dictConfig()`. Note that future enhancements to configuration functionality will be added to `dictConfig()`, so it's worth considering transitioning to this newer API when it's convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The `level` entry can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL` or `NOTSET`. For the root logger only, `NOTSET` means that all messages will be logged. Level values are `eval()`uated in the context of the logging package's namespace.

The `handlers` entry is a comma-separated list of handler names, which must appear in the `[handlers]` section. These names must appear in the `[handlers]` section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The `level` and `handlers` entries are interpreted as for the root logger, except that if a non-root logger's level is specified as `NOTSET`, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The `propagate` entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are **not** propagated to handlers up the hierarchy. The `qualname` entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The `class` entry indicates the handler's class (as determined by `eval()` in the `logging` package's namespace). The `level` is interpreted as for loggers, and `NOTSET` is taken to mean “log everything”.

The `formatter` entry indicates the key name of the formatter for this handler. If blank, a default formatter (`logging._defaultFormatter`) is used. If a name is specified, it must appear in the `[formatters]` section and have a corresponding section in the configuration file.

The `args` entry, when `eval()` uated in the context of the `logging` package's namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args= (('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
```

(suite sur la page suivante)

(suite de la page précédente)

```
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
```

Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

The `format` entry is the overall format string, and the `datefmt` entry is the `strftime()`-compatible date/time format string. If empty, the package substitutes something which is almost equivalent to specifying the date format string `'%Y-%m-%d %H:%M:%S'`. This format also specifies milliseconds, which are appended to the result of using the above format string, with a comma separator. An example time in this format is `2003-01-23 00:29:50,411`.

The `class` entry is optional. It indicates the name of the formatter's class (as a dotted module and class name.) This option is useful for instantiating a *Formatter* subclass. Subclasses of *Formatter* can present exception tracebacks in an expanded or condensed format.

Note : Due to the use of `eval()` as described above, there are potential security risks which result from using the `listen()` to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine ; see the `listen()` documentation for more information.

Voir aussi :

Module *logging* Référence d'API pour le module de journalisation.

Module *logging.handlers* Gestionnaires utiles inclus avec le module de journalisation.

16.8 logging.handlers — Logging handlers

Source code : [Lib/logging/handlers.py](#)

Important

This page contains only reference information. For tutorials, please see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

The following useful handlers are provided in the package. Note that three of the handlers (*StreamHandler*, *FileHandler* and *NullHandler*) are actually defined in the *logging* module itself, but have been documented here along with the other handlers.

16.8.1 StreamHandler

The *StreamHandler* class, located in the core *logging* package, sends logging output to streams such as *sys.stdout*, *sys.stderr* or any file-like object (or, more precisely, any object which supports *write()* and *flush()* methods).

class *logging.StreamHandler* (*stream=None*)

Returns a new instance of the *StreamHandler* class. If *stream* is specified, the instance will use it for logging output; otherwise, *sys.stderr* will be used.

emit (*record*)

If a formatter is specified, it is used to format the record. The record is then written to the stream with a terminator. If exception information is present, it is formatted using *traceback.print_exception()* and appended to the stream.

flush ()

Flushes the stream by calling its *flush()* method. Note that the *close()* method is inherited from *Handler* and so does no output, so an explicit *flush()* call may be needed at times.

Modifié dans la version 3.2 : The *StreamHandler* class now has a *terminator* attribute, default value *'\\n'*, which is used as the terminator when writing a formatted record to a stream. If you don't want this newline termination, you can set the handler instance's *terminator* attribute to the empty string. In earlier versions, the terminator was hardcoded as *'\\n'*.

16.8.2 FileHandler

The *FileHandler* class, located in the core *logging* package, sends logging output to a disk file. It inherits the output functionality from *StreamHandler*.

class *logging.FileHandler* (*filename, mode='a', encoding=None, delay=False*)

Returns a new instance of the *FileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, *'a'* is used. If *encoding* is not *None*, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely.

Modifié dans la version 3.6 : As well as string values, *Path* objects are also accepted for the *filename* argument.

close ()

Closes the file.

emit (*record*)

Outputs the record to the file.

16.8.3 NullHandler

Nouveau dans la version 3.1.

The *NullHandler* class, located in the core *logging* package, does not do any formatting or output. It is essentially a “no-op” handler for use by library developers.

class *logging.NullHandler*

Returns a new instance of the *NullHandler* class.

emit (*record*)

This method does nothing.

handle (*record*)

This method does nothing.

createLock ()

This method returns *None* for the lock, since there is no underlying I/O to which access needs to be serialized.

See library-config for more information on how to use *NullHandler*.

16.8.4 WatchedFileHandler

The `WatchedFileHandler` class, located in the `logging.handlers` module, is a `FileHandler` which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as `newsyslog` and `logrotate` which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore, `ST_INO` is not supported under Windows; `stat()` always returns zero for this value.

class `logging.handlers.WatchedFileHandler` (*filename*, *mode*='a', *encoding*=None, *delay*=False)

Returns a new instance of the `WatchedFileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, 'a' is used. If *encoding* is not None, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely.

Modifié dans la version 3.6 : As well as string values, `Path` objects are also accepted for the *filename* argument.

reopenIfNeeded()

Checks to see if the file has changed. If it has, the existing stream is flushed and closed and the file opened again, typically as a precursor to outputting the record to the file.

Nouveau dans la version 3.6.

emit (*record*)

Outputs the record to the file, but first calls `reopenIfNeeded()` to reopen the file if it has changed.

16.8.5 BaseRotatingHandler

The `BaseRotatingHandler` class, located in the `logging.handlers` module, is the base class for the rotating file handlers, `RotatingFileHandler` and `TimedRotatingFileHandler`. You should not need to instantiate this class, but it has attributes and methods you may need to override.

class `logging.handlers.BaseRotatingHandler` (*filename*, *mode*, *encoding*=None, *delay*=False)

The parameters are as for `FileHandler`. The attributes are :

namer

If this attribute is set to a callable, the `rotation_filename()` method delegates to this callable. The parameters passed to the callable are those passed to `rotation_filename()`.

Note : The namer function is called quite a few times during rollover, so it should be as simple and as fast as possible. It should also return the same output every time for a given input, otherwise the rollover behaviour may not work as expected.

Nouveau dans la version 3.3.

rotator

If this attribute is set to a callable, the `rotate()` method delegates to this callable. The parameters passed to the callable are those passed to `rotate()`.

Nouveau dans la version 3.3.

rotation_filename (*default_name*)

Modify the filename of a log file when rotating.

This is provided so that a custom filename can be provided.

The default implementation calls the “namer” attribute of the handler, if it’s callable, passing the default name to it. If the attribute isn’t callable (the default is None), the name is returned unchanged.

Paramètres `default_name` – The default name for the log file.

Nouveau dans la version 3.3.

rotate (*source*, *dest*)

When rotating, rotate the current log.

The default implementation calls the “rotator” attribute of the handler, if it’s callable, passing the source and dest arguments to it. If the attribute isn’t callable (the default is `None`), the source is simply renamed to the destination.

Paramètres

- **source** – The source filename. This is normally the base filename, e.g. “test.log”.
- **dest** – The destination filename. This is normally what the source is rotated to, e.g. “test.log.1”.

Nouveau dans la version 3.3.

The reason the attributes exist is to save you having to subclass - you can use the same callables for instances of `RotatingFileHandler` and `TimedRotatingFileHandler`. If either the namer or rotator callable raises an exception, this will be handled in the same way as any other exception during an `emit()` call, i.e. via the `handleError()` method of the handler.

If you need to make more significant changes to rotation processing, you can override the methods.

For an example, see `cookbook-rotator-namer`.

16.8.6 RotatingFileHandler

The `RotatingFileHandler` class, located in the `logging.handlers` module, supports rotation of disk log files.

class `logging.handlers.RotatingFileHandler` (*filename*, *mode*='a', *maxBytes*=0, *backupCount*=0, *encoding*=None, *delay*=False)

Returns a new instance of the `RotatingFileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, 'a' is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely.

You can use the *maxBytes* and *backupCount* values to allow the file to *rollover* at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly *maxBytes* in length; but if either of *maxBytes* or *backupCount* is zero, rollover never occurs, so you generally want to set *backupCount* to at least 1, and have a non-zero *maxBytes*. When *backupCount* is non-zero, the system will save old log files by appending the extensions “.1”, “.2” etc., to the filename. For example, with a *backupCount* of 5 and a base file name of `app.log`, you would get `app.log`, `app.log.1`, `app.log.2`, up to `app.log.5`. The file being written to is always `app.log`. When this file is filled, it is closed and renamed to `app.log.1`, and if files `app.log.1`, `app.log.2`, etc. exist, then they are renamed to `app.log.2`, `app.log.3` etc. respectively.

Modifié dans la version 3.6 : As well as string values, `Path` objects are also accepted for the *filename* argument.

doRollover ()

Does a rollover, as described above.

emit (*record*)

Outputs the record to the file, catering for rollover as described previously.

16.8.7 TimedRotatingFileHandler

The `TimedRotatingFileHandler` class, located in the `logging.handlers` module, supports rotation of disk log files at certain timed intervals.

class `logging.handlers.TimedRotatingFileHandler` (*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*)

Returns a new instance of the `TimedRotatingFileHandler` class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of *when* and *interval*.

You can use the *when* to specify the type of *interval*. The list of possible values is below. Note that they are not case sensitive.

Valeur	Type of interval	If/how <i>atTime</i> is used
'S'	Seconds	Ignoré
'M'	Minutes	Ignoré
'H'	Hours	Ignoré
'D'	Days	Ignoré
'W0' - 'W6'	Weekday (0=Monday)	Used to compute initial rollover time
'midnight'	Roll over at midnight, if <i>atTime</i> not specified, else at time <i>atTime</i>	Used to compute initial rollover time

When using weekday-based rotation, specify “W0” for Monday, “W1” for Tuesday, and so on up to “W6” for Sunday. In this case, the value passed for *interval* isn’t used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format `%Y-%m-%d_%H-%M-%S` or a leading portion thereof, depending on the rollover interval. When computing the next rollover time for the first time (when the handler is created), the last modification time of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the *utc* argument is true, times in UTC will be used; otherwise local time is used.

If *backupCount* is nonzero, at most *backupCount* files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If *delay* is true, then file opening is deferred until the first call to `emit()`.

If *atTime* is not `None`, it must be a `datetime.time` instance which specifies the time of day when rollover occurs, for the cases where rollover is set to happen « at midnight » or « on a particular weekday ». Note that in these cases, the *atTime* value is effectively used to compute the *initial* rollover, and subsequent rollovers would be calculated via the normal interval calculation.

Note : Calculation of the initial rollover time is done when the handler is initialised. Calculation of subsequent rollover times is done only when rollover occurs, and rollover occurs only when emitting output. If this is not kept in mind, it might lead to some confusion. For example, if an interval of « every minute » is set, that does not mean you will always see log files with times (in the filename) separated by a minute; if, during application execution, logging output is generated more frequently than once a minute, *then* you can expect to see log files with times separated by a minute. If, on the other hand, logging messages are only output once every five minutes (say), then there will be gaps in the file times corresponding to the minutes where no output (and hence no rollover) occurred.

Modifié dans la version 3.4 : *atTime* parameter was added.

Modifié dans la version 3.6 : As well as string values, `Path` objects are also accepted for the *filename* argument.

doRollover()

Does a rollover, as described above.

emit(record)

Outputs the record to the file, catering for rollover as described above.

16.8.8 SocketHandler

The `SocketHandler` class, located in the `logging.handlers` module, sends logging output to a network socket. The base class uses a TCP socket.

class `logging.handlers.SocketHandler` (*host*, *port*)

Returns a new instance of the `SocketHandler` class intended to communicate with a remote machine whose address is given by *host* and *port*.

Modifié dans la version 3.4 : If *port* is specified as `None`, a Unix domain socket is created using the value in *host* - otherwise, a TCP socket is created.

close ()

Closes the socket.

emit ()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a `LogRecord`, use the `makeLogRecord()` function.

handleError ()

Handles an error which has occurred during `emit()`. The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket ()

This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (`socket.SOCK_STREAM`).

makePickle (*record*)

Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket.

Note that pickles aren't completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

send (*packet*)

Send a pickled string *packet* to the socket. This function allows for partial sends which can happen when the network is busy.

createSocket ()

Tries to create a socket; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can't be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes :

- `retryStart` (initial delay, defaulting to 1.0 seconds).
- `retryFactor` (multiplier, defaulting to 2.0).
- `retryMax` (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up *after* the handler has been used, you could lose messages (since the handler won't even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).

16.8.9 DatagramHandler

The `DatagramHandler` class, located in the `logging.handlers` module, inherits from `SocketHandler` to support sending logging messages over UDP sockets.

class `logging.handlers.DatagramHandler` (*host*, *port*)

Returns a new instance of the `DatagramHandler` class intended to communicate with a remote machine whose address is given by *host* and *port*.

Modifié dans la version 3.4 : If *port* is specified as `None`, a Unix domain socket is created using the value in *host* - otherwise, a UDP socket is created.

emit ()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a `LogRecord`, use the `makeLogRecord()` function.

makeSocket ()

The factory method of `SocketHandler` is here overridden to create a UDP socket (`socket.SOCK_DGRAM`).

send (*s*)

Send a pickled string to a socket.

16.8.10 SysLogHandler

The `SysLogHandler` class, located in the `logging.handlers` module, supports sending logging messages to a remote or local Unix syslog.

class `logging.handlers.SysLogHandler` (*address*=(`'localhost'`, `SYSLOG_UDP_PORT`), *facility*=`LOG_USER`, *sockettype*=`socket.SOCK_DGRAM`)

Returns a new instance of the `SysLogHandler` class intended to communicate with a remote Unix machine whose address is given by *address* in the form of a (*host*, *port*) tuple. If *address* is not specified, (`'localhost'`, `514`) is used. The address is used to open a socket. An alternative to providing a (*host*, *port*) tuple is providing an address as a string, for example `"/dev/log"`. In this case, a Unix domain socket is used to send the message to the syslog. If *facility* is not specified, `LOG_USER` is used. The type of socket opened depends on the *sockettype* argument, which defaults to `socket.SOCK_DGRAM` and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of `socket.SOCK_STREAM`. Note that if your server is not listening on UDP port 514, `SysLogHandler` may appear not to work. In that case, check what address you should be using for a domain socket - it's system dependent. For example, on Linux it's usually `"/dev/log"` but on OS/X it's `"/var/run/syslog"`. You'll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.

Modifié dans la version 3.2 : *sockettype* was added.

close ()

Closes the socket to the remote host.

emit (*record*)

The record is formatted, and then sent to the syslog server. If exception information is present, it is *not* sent to the server.

Modifié dans la version 3.2.1 : (See : [bpo-12168](#).) In earlier versions, the message sent to the syslog daemons was always terminated with a NUL byte, because early versions of these daemons expected a NUL terminated message - even though it's not in the relevant specification ([RFC 5424](#)). More recent versions of these daemons don't expect the NUL byte but strip it off if it's there, and even more recent daemons (which adhere more closely to RFC 5424) pass the NUL byte on as part of the message.

To enable easier handling of syslog messages in the face of all these differing daemon behaviours, the appending of the NUL byte has been made configurable, through the use of a class-level attribute, `append_nul`.

This defaults to `True` (preserving the existing behaviour) but can be set to `False` on a `SysLogHandler` instance in order for that instance to *not* append the NUL terminator.

Modifié dans la version 3.3 : (See : [bpo-12419](#).) In earlier versions, there was no facility for an « ident » or « tag » prefix to identify the source of the message. This can now be specified using a class-level attribute, defaulting to `" "` to preserve existing behaviour, but which can be overridden on a `SysLogHandler` instance in order for that instance to prepend the ident to every message handled. Note that the provided ident must be text, not bytes, and is prepended to the message exactly as is.

encodePriority (*facility, priority*)

Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic `LOG_` values are defined in `SysLogHandler` and mirror the values defined in the `sys/syslog.h` header file.

Priorities

Name (string)	Symbolic value
alert	LOG_ALERT
crit ou critical	LOG_CRIT
debug	LOG_DEBUG
emerg ou panic	LOG_EMERG
err ou error	LOG_ERR
info	LOG_INFO
notice	LOG_NOTICE
warn ou warning	LOG_WARNING

Facilities

Name (string)	Symbolic value
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority (*levelname*)

Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL` to the equivalent syslog names, and all other level names to “warning”.

16.8.11 NTEventLogHandler

The `NTEventLogHandler` class, located in the `logging.handlers` module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond's Win32 extensions for Python installed.

class `logging.handlers.NTEventLogHandler` (*appname*, *dllname=None*, *logtype='Application'*)

Returns a new instance of the `NTEventLogHandler` class. The *appname* is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The *dllname* should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, 'win32service.pyd' is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The *logtype* is one of 'Application', 'System' or 'Security', and defaults to 'Application'.

close ()

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

emit (*record*)

Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory (*record*)

Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

getEventType (*record*)

Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler's *typemap* attribute, which is set up in `__init__()` to a dictionary which contains mappings for DEBUG, INFO, WARNING, ERROR and CRITICAL. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler's *typemap* attribute.

getMessageID (*record*)

Returns the message ID for the record. If you are using your own messages, you could do this by having the *msg* passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in `win32service.pyd`.

16.8.12 SMTPHandler

The `SMTPHandler` class, located in the `logging.handlers` module, supports sending logging messages to an email address via SMTP.

class `logging.handlers.SMTPHandler` (*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=1.0*)

Returns a new instance of the `SMTPHandler` class. The instance is initialized with the from and to addresses and subject line of the email. The *toaddrs* should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the *mailhost* argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the *credentials* argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the *secure* argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the `smtpplib.SMTP.starttls()` method.)

A timeout can be specified for communication with the SMTP server using the *timeout* argument.

Nouveau dans la version 3.3 : The *timeout* argument was added.

emit (*record*)

Formats the record and sends it to the specified addressees.

getSubject (*record*)

If you want to specify a subject line which is record-dependent, override this method.

16.8.13 MemoryHandler

The *MemoryHandler* class, located in the *logging.handlers* module, supports buffering of logging records in memory, periodically flushing them to a *target* handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

MemoryHandler is a subclass of the more general *BufferingHandler*, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling *shouldFlush()* to see if the buffer should be flushed. If it should, then *flush()* is expected to do the flushing.

class *logging.handlers.BufferingHandler* (*capacity*)

Initializes the handler with a buffer of the specified capacity.

emit (*record*)

Appends the record to the buffer. If *shouldFlush()* returns true, calls *flush()* to process the buffer.

flush ()

You can override this to implement custom flushing behavior. This version just zaps the buffer to empty.

shouldFlush (*record*)

Returns true if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

class *logging.handlers.MemoryHandler* (*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)

Returns a new instance of the *MemoryHandler* class. The instance is initialized with a buffer size of *capacity*. If *flushLevel* is not specified, *ERROR* is used. If no *target* is specified, the target will need to be set using *setTarget()* before this handler does anything useful. If *flushOnClose* is specified as *False*, then the buffer is *not* flushed when the handler is closed. If not specified or specified as *True*, the previous behaviour of flushing the buffer will occur when the handler is closed.

Modifié dans la version 3.6 : The *flushOnClose* parameter was added.

close ()

Calls *flush()*, sets the target to *None* and clears the buffer.

flush ()

For a *MemoryHandler*, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when this happens. Override if you want different behavior.

setTarget (*target*)

Sets the target handler for this handler.

shouldFlush (*record*)

Checks for buffer full or a record at the *flushLevel* or higher.

16.8.14 HTTPHandler

The *HTTPHandler* class, located in the *logging.handlers* module, supports sending logging messages to a Web server, using either GET or POST semantics.

class *logging.handlers.HTTPHandler* (*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*)

Returns a new instance of the *HTTPHandler* class. The *host* can be of the form *host:port*, should you need to use a specific port number. If no *method* is specified, GET is used. If *secure* is true, a HTTPS connection will be used. The *context* parameter may be set to a *ssl.SSLContext* instance to configure the SSL settings used for the HTTPS connection. If *credentials* is specified, it should be a 2-tuple consisting of userid and password, which will be placed in a HTTP “Authorization” header using Basic authentication. If you specify credentials, you should also specify *secure=True* so that your userid and password are not passed in cleartext across the wire.

Modifié dans la version 3.5 : The *context* parameter was added.

mapLogRecord (*record*)

Provides a dictionary, based on *record*, which is to be URL-encoded and sent to the web server. The default implementation just returns *record.__dict__*. This method can be overridden if e.g. only a subset of *LogRecord* is to be sent to the web server, or if more specific customization of what's sent to the server is required.

emit (*record*)

Sends the record to the Web server as a URL-encoded dictionary. The *mapLogRecord()* method is used to convert the record to the dictionary to be sent.

Note : Since preparing a record for sending it to a Web server is not the same as a generic formatting operation, using *setFormatter()* to specify a *Formatter* for a *HTTPHandler* has no effect. Instead of calling *format()*, this handler calls *mapLogRecord()* and then *urllib.parse.urlencode()* to encode the dictionary in a form suitable for sending to a Web server.

16.8.15 QueueHandler

Nouveau dans la version 3.2.

The *QueueHandler* class, located in the *logging.handlers* module, supports sending logging messages to a queue, such as those implemented in the *queue* or *multiprocessing* modules.

Along with the *QueueListener* class, *QueueHandler* can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in Web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via *SMTPHandler*) are done on a separate thread.

class logging.handlers.**QueueHandler** (*queue*)

Returns a new instance of the *QueueHandler* class. The instance is initialized with the queue to send messages to. The queue can be any queue-like object ; it's used as-is by the *enqueue()* method, which needs to know how to send messages to it.

emit (*record*)

Enqueues the result of preparing the LogRecord.

prepare (*record*)

Prepares a record for queuing. The object returned by this method is enqueued.

The base implementation formats the record to merge the message and arguments, and removes unpickleable items from the record in-place.

You might want to override this method if you want to convert the record to a dict or JSON string, or send a modified copy of the record while leaving the original intact.

enqueue (*record*)

Enqueues the record on the queue using *put_nowait()* ; you may want to override this if you want to use blocking behaviour, or a timeout, or a customized queue implementation.

16.8.16 QueueListener

Nouveau dans la version 3.2.

The `QueueListener` class, located in the `logging.handlers` module, supports receiving logging messages from a queue, such as those implemented in the `queue` or `multiprocessing` modules. The messages are received from a queue in an internal thread and passed, on the same thread, to one or more handlers for processing. While `QueueListener` is not itself a handler, it is documented here because it works hand-in-hand with `QueueHandler`.

Along with the `QueueHandler` class, `QueueListener` can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in Web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via `SMTPHandler`) are done on a separate thread.

class `logging.handlers.QueueListener` (*queue*, **handlers*, *respect_handler_level=False*)

Returns a new instance of the `QueueListener` class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it's passed as-is to the `dequeue()` method, which needs to know how to get messages from it. If `respect_handler_level` is True, a handler's level is respected (compared with the level for the message) when deciding whether to pass messages to that handler; otherwise, the behaviour is as in previous Python versions - to always pass each message to each handler.

Modifié dans la version 3.5 : The `respect_handler_levels` argument was added.

dequeue (*block*)

Dequeues a record and return it, optionally blocking.

The base implementation uses `get()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

prepare (*record*)

Prepare a record for handling.

This implementation just returns the passed-in record. You may want to override this method if you need to do any custom marshalling or manipulation of the record before passing it to the handlers.

handle (*record*)

Handle a record.

This just loops through the handlers offering them the record to handle. The actual object passed to the handlers is that which is returned from `prepare()`.

start ()

Starts the listener.

This starts up a background thread to monitor the queue for LogRecords to process.

stop ()

Stops the listener.

This asks the thread to terminate, and then waits for it to do so. Note that if you don't call this before your application exits, there may be some records still left on the queue, which won't be processed.

enqueue_sentinel ()

Writes a sentinel to the queue to tell the listener to quit. This implementation uses `put_nowait()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

Nouveau dans la version 3.3.

Voir aussi :

Module `logging` Référence d'API pour le module de journalisation.

Module `logging.config` API de configuration pour le module de journalisation.

16.9 Saisie de mot de passe portable

Source code : [Lib/getpass.py](#)

Le module `getpass` fournit 2 fonctions :

`getpass.getpass` (*prompt*=`'Password : '`, *stream*=`None`)

Affiche une demande de mot de passe sans renvoyer d'écho. L'utilisateur est invité en utilisant la string *prompt*, avec en valeur par défaut `'Password: '`. Avec Unix, l'invite est écrite dans l'objet fichier *stream* en utilisant si besoin le *replace error handler*. *stream* sera par défaut le terminal de contrôle (`/dev/tty`), ou si celui ci n'est pas disponible ce sera `sys.stderr` (cet argument sera ignoré sur Windows).

Si aucune saisie en mode sans affichage n'est disponible, `getpass()` se résoudra à afficher un message d'avertissement vers *stream*, puis lire l'entrée depuis `sys.stdin`, en levant une `GetPassWarning`.

Note : Si vous appelez `getpass` depuis IDLE, la saisie peut être faite dans le terminal depuis lequel IDLE a été lancé, plutôt que dans la fenêtre d'IDLE.

exception `getpass.GetPassWarning`

Une sous classe d'exception `UserWarning` est levée quand le mot de passe saisi pourrait être affiché.

`getpass.getuser` ()

Renvoie le *login name* de l'utilisateur.

Cette fonction examine les variables d'environnement `LOGNAME`, `USER`, `LNAME` et `USERNAME`, dans cet ordre, et renvoie la valeur de la première qui a comme valeur une string non vide. Si aucune des variables n'est renseignée, dans le cas de systèmes qui prennent en charge le module `pwd`, le *login name* de la base de données des mots de passes est renvoyé, pour les autres systèmes une exception est levée.

En général, préférez cette fonction à `os.getlogin()`.

16.10 curses — Terminal handling for character-cell displays

The `curses` module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the Unix environment, versions are available for Windows, DOS, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

Note : Whenever the documentation mentions a *character* it can be specified as an integer, a one-character Unicode string or a one-byte byte string.

Whenever the documentation mentions a *character string* it can be specified as a Unicode string or a byte string.

Note : Since version 5.4, the ncurses library decides how to interpret non-ASCII data using the `nl_langinfo` function. That means that you have to call `locale.setlocale()` in the application and encode Unicode strings using one of the system's available encodings. This example uses the system's default encoding :


```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

Then use *code* as the encoding for *str.encode()* calls.

Voir aussi :

Module `curses.ascii` Utilities for working with ASCII characters, regardless of your locale settings.

Module `curses.panel` A panel stack extension that adds depth to curses windows.

Module `curses.textpad` Editable text widget for curses supporting **Emacs**-like bindings.

curses-howto Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

The `Tools/demo/` directory in the Python source distribution contains some example programs using the curses bindings provided by this module.

16.10.1 Fonctions

The module `curses` defines the following exception :

exception `curses.error`

Exception raised when a curses library function returns an error.

Note : Whenever *x* or *y* arguments to a function or a method are optional, they default to the current cursor location. Whenever *attr* is optional, it defaults to `A_NORMAL`.

The module `curses` defines the following functions :

`curses.baudrate()`

Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

`curses.beep()`

Emit a short attention sound.

`curses.can_change_color()`

Return `True` or `False`, depending on whether the programmer can change the colors displayed by the terminal.

`curses.cbreak()`

Enter cbreak mode. In cbreak mode (sometimes called « rare » mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first `raw()` then `cbreak()` leaves the terminal in cbreak mode.

`curses.color_content(color_number)`

Return the intensity of the red, green, and blue (RGB) components in the color *color_number*, which must be between 0 and `COLORS`. Return a 3-tuple, containing the R,G,B values for the given color, which will be between 0 (no component) and 1000 (maximum amount of component).

`curses.color_pair(color_number)`

Return the attribute value for displaying text in the specified color. This attribute value can be combined with `A_STANDOUT`, `A_REVERSE`, and the other `A_*` attributes. `pair_number()` is the counterpart to this function.

`curses.curs_set(visibility)`

Set the cursor state. *visibility* can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, return the previous cursor state; otherwise raise an exception. On many terminals, the « visible » mode is an underline cursor and the « very visible » mode is a block cursor.

`curses.def_prog_mode()`

Save the current terminal mode as the « program » mode, the mode when the running program is using curses. (Its counterpart is the « shell » mode, for when the program is not in curses.) Subsequent calls to `reset_prog_mode()` will restore this mode.

`curses.def_shell_mode()`

Save the current terminal mode as the « shell » mode, the mode when the running program is not using curses. (Its counterpart is the « program » mode, when the program is using curses capabilities.) Subsequent calls to `reset_shell_mode()` will restore this mode.

`curses.delay_output(ms)`

Insert an *ms* millisecond pause in output.

`curses.doupdate()`

Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The `doupdate()` ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a `noutrefresh()` call after write operations such as `addstr()` have been performed on a window. The normal `refresh()` call is simply `noutrefresh()` followed by `doupdate()`; if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing `noutrefresh()` calls on all windows, followed by a single `doupdate()`.

`curses.echo()`

Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

`curses.endwin()`

De-initialize the library, and return terminal to normal status.

`curses.erasechar()`

Return the user's current erase character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

`curses.filter()`

The `filter()` routine, if used, must be called before `initscr()` is called. The effect is that, during those calls, `LINES` is set to 1; the capabilities `clear`, `cup`, `cud`, `cud1`, `cuu1`, `cuu`, `vpa` are disabled; and the `home` string is set to the value of `cr`. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

`curses.flash()`

Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as “visible bell” to the audible attention signal produced by `beep()`.

`curses.flushinp()`

Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

`curses.getmouse()`

After `getch()` returns `KEY_MOUSE` to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple (*id*, *x*, *y*, *z*, *bstate*). *id* is an ID value used to distinguish multiple devices, and *x*, *y*, *z* are the event's coordinates. (*z* is currently unused.) *bstate* is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where *n* is the button number from 1 to 4 : `BUTTONn_PRESSED`, `BUTTONn_RELEASED`, `BUTTONn_CLICKED`, `BUTTONn_DOUBLE_CLICKED`, `BUTTONn_TRIPLE_CLICKED`, `BUTTON_SHIFT`, `BUTTON_CTRL`, `BUTTON_ALT`.

`curses.getsyx()`
Return the current coordinates of the virtual screen cursor as a tuple `(y, x)`. If `leaveok` is currently `True`, then return `(-1, -1)`.

`curses.getwin(file)`
Read window related data stored in the file by an earlier `putwin()` call. The routine then creates and initializes a new window using that data, returning the new window object.

`curses.has_colors()`
Return `True` if the terminal can display colors; otherwise, return `False`.

`curses.has_ic()`
Return `True` if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.has_il()`
Return `True` if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.has_key(ch)`
Take a key value `ch`, and return `True` if the current terminal type recognizes a key with that value.

`curses.halfdelay(tenths)`
Used for half-delay mode, which is similar to `cbreak` mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, raise an exception if nothing has been typed. The value of *tenths* must be a number between 1 and 255. Use `nocbreak()` to leave half-delay mode.

`curses.init_color(color_number, r, g, b)`
Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of *color_number* must be between 0 and `COLORS`. Each of *r*, *g*, *b*, must be a value between 0 and 1000. When `init_color()` is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if `can_change_color()` returns `True`.

`curses.init_pair(pair_number, fg, bg)`
Change the definition of a color-pair. It takes three arguments : the number of the color-pair to be changed, the foreground color number, and the background color number. The value of *pair_number* must be between 1 and `COLOR_PAIRS - 1` (the 0 color pair is wired to white on black and cannot be changed). The value of *fg* and *bg* arguments must be between 0 and `COLORS`. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

`curses.initscr()`
Initialize the library. Return a *window* object which represents the whole screen.

Note : If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

`curses.is_term_resized(nlines, ncols)`
Return `True` if `resize_term()` would modify the window structure, `False` otherwise.

`curses.isendwin()`
Return `True` if `endwin()` has been called (that is, the curses library has been deinitialized).

`curses.keyname(k)`
Return the name of the key numbered *k* as a bytes object. The name of a key generating printable ASCII character is the key's character. The name of a control-key combination is a two-byte bytes object consisting of a caret (`b'^'`) followed by the corresponding printable ASCII character. The name of an alt-key combination (128–255) is a bytes object consisting of the prefix `b'M-'` followed by the name of the corresponding ASCII character.

`curses.killchar()`

Return the user's current line kill character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

`curses.longname()`

Return a bytes object containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to `initscr()`.

`curses.meta(flag)`

If *flag* is `True`, allow 8-bit characters to be input. If *flag* is `False`, allow only 7-bit chars.

`curses.mouseinterval(interval)`

Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 msec, or one fifth of a second.

`curses.mousemask(mousemask)`

Set the mouse events to be reported, and return a tuple (*availmask*, *oldmask*). *availmask* indicates which of the specified mouse events can be reported; on complete failure it returns 0. *oldmask* is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

`curses.napms(ms)`

Sleep for *ms* milliseconds.

`curses.newpad(nlines, ncols)`

Create and return a pointer to a new pad data structure with the given number of lines and columns. Return a pad as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The `refresh()` and `noutrefresh()` methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*; the *p* arguments refer to the upper left corner of the pad region to be displayed and the *s* arguments define a clipping box on the screen within which the pad region is to be displayed.

`curses.newwin(nlines, ncols)`

`curses.newwin(nlines, ncols, begin_y, begin_x)`

Return a new *window*, whose left-upper corner is at (*begin_y*, *begin_x*), and whose height/width is *nlines/ncols*.

By default, the window will extend from the specified position to the lower right corner of the screen.

`curses.nl()`

Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

`curses.nocbreak()`

Leave cbreak mode. Return to normal « cooked » mode with line buffering.

`curses.noecho()`

Leave echo mode. Echoing of input characters is turned off.

`curses.nonl()`

Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch('\n')`, which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

`curses.noqiflush()`

When the `noqiflush()` routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done. You may want to call `noqiflush()` in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

`curses.noraw()`
 Leave raw mode. Return to normal « cooked » mode with line buffering.

`curses.pair_content(pair_number)`
 Return a tuple (fg, bg) containing the colors for the requested color pair. The value of *pair_number* must be between 1 and `COLOR_PAIRS - 1`.

`curses.pair_number(attr)`
 Return the number of the color-pair set by the attribute value *attr*. `color_pair()` is the counterpart to this function.

`curses.putp(str)`
 Equivalent to `tputs(str, 1, putchar)`; emit the value of a specified terminfo capability for the current terminal. Note that the output of `putp()` always goes to standard output.

`curses.qiflush([flag])`
 If *flag* is False, the effect is the same as calling `noqiflush()`. If *flag* is True, or no argument is provided, the queues will be flushed when these control characters are read.

`curses.raw()`
 Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

`curses.reset_prog_mode()`
 Restore the terminal to « program » mode, as previously saved by `def_prog_mode()`.

`curses.reset_shell_mode()`
 Restore the terminal to « shell » mode, as previously saved by `def_shell_mode()`.

`curses.resetty()`
 Restore the state of the terminal modes to what it was at the last call to `savetty()`.

`curses.resize_term(nlines, ncols)`
 Backend function used by `resizeterm()`, performing most of the work; when resizing the windows, `resize_term()` blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The `resize_term()` function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

`curses.resizeterm(nlines, ncols)`
 Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

`curses.savetty()`
 Save the current state of the terminal modes in a buffer, usable by `resetty()`.

`curses.setsyx(y, x)`
 Set the virtual screen cursor to y, x. If y and x are both -1, then `leaveok` is set True.

`curses.setupterm(term=None, fd=-1)`
 Initialize the terminal. *term* is a string giving the terminal name, or None; if omitted or None, the value of the TERM environment variable will be used. *fd* is the file descriptor to which any initialization sequences will be sent; if not supplied or -1, the file descriptor for `sys.stdout` will be used.

`curses.start_color()`
 Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after `initscr()`.
`start_color()` initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the `curses` module, `COLORS` and `COLOR_PAIRS`, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

`curses.termattrs()`

Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

`curses.termname()`

Return the value of the environment variable `TERM`, as a bytes object, truncated to 14 characters.

`curses.tigetflag(capname)`

Return the value of the Boolean capability corresponding to the terminfo capability name *capname* as an integer. Return the value `-1` if *capname* is not a Boolean capability, or `0` if it is canceled or absent from the terminal description.

`curses.tigetnum(capname)`

Return the value of the numeric capability corresponding to the terminfo capability name *capname* as an integer. Return the value `-2` if *capname* is not a numeric capability, or `-1` if it is canceled or absent from the terminal description.

`curses.tigetstr(capname)`

Return the value of the string capability corresponding to the terminfo capability name *capname* as a bytes object. Return `None` if *capname* is not a terminfo « string capability », or is canceled or absent from the terminal description.

`curses.tparm(str[, ...])`

Instantiate the bytes object *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. `tparm(tigetstr("cup"), 5, 3)` could result in `b'\033[6;4H'`, the exact result depending on terminal type.

`curses.typeahead(fd)`

Specify that the file descriptor *fd* be used for typeahead checking. If *fd* is `-1`, then no typeahead checking is done. The curses library does « line-breakout optimization » by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until `refresh` or `doupdate` is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

`curses.unctrl(ch)`

Return a bytes object which is a printable representation of the character *ch*. Control characters are represented as a caret followed by the character, for example as `b'^C'`. Printing characters are left as they are.

`curses.ungetch(ch)`

Push *ch* so the next `getch()` will return it.

Note : Only one *ch* can be pushed before `getch()` is called.

`curses.update_lines_cols()`

Update `LINES` and `COLS`. Useful for detecting manual screen resize.
Nouveau dans la version 3.5.

`curses.unget_wch(ch)`

Push *ch* so the next `get_wch()` will return it.

Note : Only one *ch* can be pushed before `get_wch()` is called.

Nouveau dans la version 3.3.

`curses.ungetmouse(id, x, y, z, bstate)`

Push a `KEY_MOUSE` event onto the input queue, associating the given state data with it.

`curses.use_env(flag)`

If used, this function should be called before `initscr()` or `newterm` are called. When *flag* is `False`, the values of lines and columns specified in the terminfo database will be used, even if environment variables `LINES` and `COLUMNS` (used by default) are set, or if `curses` is running in a window (in which case default behavior would be to use the window size if `LINES` and `COLUMNS` are not set).

`curses.use_default_colors()`

Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number `-1`. After calling this function, `init_pair(x, curses.COLOR_RED, -1)` initializes, for instance, color pair *x* to a red foreground color on the default background.

`curses.wrapper(func, ...)`

Initialize `curses` and call another callable object, *func*, which should be the rest of your `curses`-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object *func* is then passed the main window “`stdscr`” as its first argument, followed by any other arguments passed to `wrapper()`. Before calling *func*, `wrapper()` turns on `cbreak` mode, turns off `echo`, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on `echo`, and disables the terminal keypad.

16.10.2 Window Objects

Window objects, as returned by `initscr()` and `newwin()` above, have the following methods and attributes :

`window.addch(ch[, attr])`

`window.addch(y, x, ch[, attr])`

Paint character *ch* at (*y*, *x*) with attributes *attr*, overwriting any character previously painter at that location. By default, the character position and attributes are the current settings for the window object.

Note : Writing outside the window, subwindow, or pad raises a `curses.error`. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

`window.addnstr(str, n[, attr])`

`window.addnstr(y, x, str, n[, attr])`

Paint at most *n* characters of the character string *str* at (*y*, *x*) with attributes *attr*, overwriting anything previously on the display.

`window.addstr(str[, attr])`

`window.addstr(y, x, str[, attr])`

Paint the character string *str* at (*y*, *x*) with attributes *attr*, overwriting anything previously on the display.

Note : Writing outside the window, subwindow, or pad raises `curses.error`. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.

`window.attroff(attr)`

Remove attribute *attr* from the « background » set applied to all writes to the current window.

`window.atttron(attr)`

Add attribute *attr* from the « background » set applied to all writes to the current window.

`window.attrset(attr)`

Set the « background » set of attributes to *attr*. This set is initially 0 (no attributes).

`window.bkgd(ch[, attr])`

Set the background property of the window to the character *ch*, with attributes *attr*. The change is then applied to

every character position in that window :

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

`window.bkgdset (ch[, attr])`

Set the window's background. A window's background consists of a character and any combination of attributes. The attribute part of the background is combined (OR'ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

`window.border ([ls[, rs[, ts[, bs[, tl[, tr[, bl[, br]]]]]]]])`

Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details.

Note : A 0 value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table :

Paramètre	Description	Valeur par défaut
<i>ls</i>	Left side	ACS_VLINE
<i>rs</i>	Right side	ACS_VLINE
<i>ts</i>	Top	ACS_HLINE
<i>bs</i>	Bottom	ACS_HLINE
<i>tl</i>	Upper-left corner	ACS_ULCORNER
<i>tr</i>	Upper-right corner	ACS_URCORNER
<i>bl</i>	Bottom-left corner	ACS_LLCORNER
<i>br</i>	Bottom-right corner	ACS_LRCORNER

`window.box ([vertch, horch])`

Similar to `border()`, but both *ls* and *rs* are *vertch* and both *ts* and *bs* are *horch*. The default corner characters are always used by this function.

`window.chgat (attr)`

`window.chgat (num, attr)`

`window.chgat (y, x, attr)`

`window.chgat (y, x, num, attr)`

Set the attributes of *num* characters at the current cursor position, or at position (*y*, *x*) if supplied. If *num* is not given or is -1, the attribute will be set on all the characters to the end of the line. This function moves cursor to position (*y*, *x*) if supplied. The changed line will be touched using the `touchline()` method so that the contents will be redisplayed by the next window refresh.

`window.clear()`

Like `erase()`, but also cause the whole window to be repainted upon next call to `refresh()`.

`window.clearok (flag)`

If *flag* is True, the next call to `refresh()` will clear the window completely.

`window.clrtoeol()`

Erase from cursor to the end of the window : all lines below the cursor are deleted, and then the equivalent of `clrtoeol()` is performed.

`window.clrtoeol()`

Erase from cursor to the end of the line.

`window.cursyncup()`

Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

`window.delch([y, x])`
Delete any character at (y, x) .

`window.deleteln()`
Delete the line under the cursor. All following lines are moved up by one line.

`window.derwin(begin_y, begin_x)`
`window.derwin(nlines, ncols, begin_y, begin_x)`
An abbreviation for « derive window », `derwin()` is the same as calling `subwin()`, except that `begin_y` and `begin_x` are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

`window.echochar(ch[, attr])`
Add character `ch` with attribute `attr`, and immediately call `refresh()` on the window.

`window.enclose(y, x)`
Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning True or False. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

`window.encoding`
Encoding used to encode method arguments (Unicode strings and characters). The encoding attribute is inherited from the parent window when a subwindow is created, for example with `window.subwin()`. By default, the locale encoding is used (see `locale.getpreferredencoding()`).
Nouveau dans la version 3.3.

`window.erase()`
Clear the window.

`window.getbegyx()`
Return a tuple (y, x) of co-ordinates of upper-left corner.

`window.getbkgd()`
Return the given window's current background character/attribute pair.

`window.getch([y, x])`
Get a character. Note that the integer returned does *not* have to be in ASCII range : function keys, keypad keys and so on are represented by numbers higher than 255. In no-delay mode, return -1 if there is no input, otherwise wait until a key is pressed.

`window.get_wch([y, x])`
Get a wide character. Return a character for most keys, or an integer for function keys, keypad keys, and other special keys. In no-delay mode, raise an exception if there is no input.
Nouveau dans la version 3.3.

`window.getkey([y, x])`
Get a character, returning a string instead of an integer, as `getch()` does. Function keys, keypad keys and other special keys return a multibyte string containing the key name. In no-delay mode, raise an exception if there is no input.

`window.getmaxyx()`
Return a tuple (y, x) of the height and width of the window.

`window.getparyx()`
Return the beginning coordinates of this window relative to its parent window as a tuple (y, x) . Return $(-1, -1)$ if this window has no parent.

`window.getstr()`
`window.getstr(n)`
`window.getstr(y, x)`

`window.getstr(y, x, n)`
 Read a bytes object from the user, with primitive line editing capacity.

`window.getyx()`
 Return a tuple `(y, x)` of current cursor position relative to the window's upper-left corner.

`window.hline(ch, n)`
`window.hline(y, x, ch, n)`
 Display a horizontal line starting at `(y, x)` with length `n` consisting of the character `ch`.

`window.idcok(flag)`
 If `flag` is `False`, curses no longer considers using the hardware insert/delete character feature of the terminal; if `flag` is `True`, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

`window.idlok(flag)`
 If `flag` is `True`, `curses` will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

`window.immedok(flag)`
 If `flag` is `True`, any change in the window image automatically causes the window to be refreshed; you no longer have to call `refresh()` yourself. However, it may degrade performance considerably, due to repeated calls to `wrefresh`. This option is disabled by default.

`window.inch([y, x])`
 Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

`window.insch(ch[, attr])`
`window.insch(y, x, ch[, attr])`
 Paint character `ch` at `(y, x)` with attributes `attr`, moving the line from position `x` right by one character.

`window.insdelln(nlines)`
 Insert `nlines` lines into the specified window above the current line. The `nlines` bottom lines are lost. For negative `nlines`, delete `nlines` lines starting with the one under the cursor, and move the remaining lines up. The bottom `nlines` lines are cleared. The current cursor position remains the same.

`window.insertln()`
 Insert a blank line under the cursor. All following lines are moved down by one line.

`window.insnstr(str, n[, attr])`
`window.insnstr(y, x, str, n[, attr])`
 Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to `n` characters. If `n` is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to `y, x`, if specified).

`window.insstr(str[, attr])`
`window.insstr(y, x, str[, attr])`
 Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to `y, x`, if specified).

`window.instr([n])`
`window.instr(y, x[, n])`
 Return a bytes object of characters, extracted from the window starting at the current cursor position, or at `y, x` if specified. Attributes are stripped from the characters. If `n` is specified, `instr()` returns a string at most `n` characters long (exclusive of the trailing NUL).

`window.is_linetouched (line)`

Return True if the specified line was modified since the last call to `refresh()`; otherwise return False. Raise a `curses.error` exception if `line` is not valid for the given window.

`window.is_wintouched ()`

Return True if the specified window was modified since the last call to `refresh()`; otherwise return False.

`window.keypad (flag)`

If `flag` is True, escape sequences generated by some keys (keypad, function keys) will be interpreted by `curses`.

If `flag` is False, escape sequences will be left as is in the input stream.

`window.leaveok (flag)`

If `flag` is True, cursor is left where it is on update, instead of being at « cursor position. » This reduces cursor movement where possible. If possible the cursor will be made invisible.

If `flag` is False, cursor will always be at « cursor position » after an update.

`window.move (new_y, new_x)`

Move cursor to `(new_y, new_x)`.

`window.mvderwin (y, x)`

Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

`window.mvwin (new_y, new_x)`

Move the window so its upper-left corner is at `(new_y, new_x)`.

`window.nodelay (flag)`

If `flag` is True, `getch()` will be non-blocking.

`window.notimeout (flag)`

If `flag` is True, escape sequences will not be timed out.

If `flag` is False, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

`window.noutrefresh ()`

Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call `doupdate()`.

`window.overlay (destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])`

Overlay the window on top of `destwin`. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of `destwin`.

To get fine-grained control over the copied region, the second form of `overlay()` can be used. `sminrow` and `smincol` are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

`window.overwrite (destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])`

Overwrite the window on top of `destwin`. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of `destwin`.

To get fine-grained control over the copied region, the second form of `overwrite()` can be used. `sminrow` and `smincol` are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

`window.putwin (file)`

Write all data associated with the window into the provided file object. This information can be later retrieved using the `getwin()` function.

`window.redrawln(beg, num)`

Indicate that the *num* screen lines, starting at line *beg*, are corrupted and should be completely redrawn on the next `refresh()` call.

`window.redrawwin()`

Touch the entire window, causing it to be completely redrawn on the next `refresh()` call.

`window.refresh([pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol])`

Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with `newpad()`. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

`window.resize(nlines, ncols)`

Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window's data is filled with blanks that have the current background rendition (as set by `bkgdset()`) merged into them.

`window.scroll([lines=1])`

Scroll the screen or scrolling region upward by *lines* lines.

`window.scrollok(flag)`

Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If *flag* is `False`, the cursor is left on the bottom line. If *flag* is `True`, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call `idlok()`.

`window.setscreg(top, bottom)`

Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

`window.standend()`

Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

`window.standout()`

Turn on attribute `A_STANDOUT`.

`window.subpad(begin_y, begin_x)`

`window.subpad(nlines, ncols, begin_y, begin_x)`

Return a sub-window, whose upper-left corner is at (*begin_y*, *begin_x*), and whose width/height is *ncols/nlines*.

`window.subwin(begin_y, begin_x)`

`window.subwin(nlines, ncols, begin_y, begin_x)`

Return a sub-window, whose upper-left corner is at (*begin_y*, *begin_x*), and whose width/height is *ncols/nlines*.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

`window.syncdown()`

Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by `refresh()`, so it should almost never be necessary to call it manually.

`window.syncok(flag)`

If *flag* is `True`, then `syncup()` is called automatically whenever there is a change in the window.

`window.syncup()`

Touch all locations in ancestors of the window that have been changed in the window.

`window.timeout(delay)`

Set blocking or non-blocking read behavior for the window. If *delay* is negative, blocking read is used (which will wait indefinitely for input). If *delay* is zero, then non-blocking read is used, and `getch()` will return `-1` if no input is waiting. If *delay* is positive, then `getch()` will block for *delay* milliseconds, and return `-1` if there is still no input at the end of that time.

`window.touchline(start, count[, changed])`

Pretend *count* lines have been changed, starting with line *start*. If *changed* is supplied, it specifies whether the affected lines are marked as having been changed (*changed*=`True`) or unchanged (*changed*=`False`).

`window.touchwin()`

Pretend the whole window has been changed, for purposes of drawing optimizations.

`window.untouchwin()`

Mark all lines in the window as unchanged since the last call to `refresh()`.

`window.vline(ch, n)`

`window.vline(y, x, ch, n)`

Display a vertical line starting at (*y*, *x*) with length *n* consisting of the character *ch*.

16.10.3 Constantes

The `curses` module defines the following data members :

`curses.ERR`

Some curses routines that return an integer, such as `getch()`, return `ERR` upon failure.

`curses.OK`

Some curses routines that return an integer, such as `napms()`, return `OK` upon success.

`curses.version`

A bytes object representing the current version of the module. Also available as `__version__`.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

Attribut	Signification
<code>A_ALTCHARSET</code>	Alternate character set mode
<code>A_BLINK</code>	Blink mode
<code>A_BOLD</code>	Bold mode
<code>A_DIM</code>	Dim mode
<code>A_INVIS</code>	Invisible or blank mode
<code>A_NORMAL</code>	Attribut normal
<code>A_PROTECT</code>	Protected mode
<code>A_REVERSE</code>	Reverse background and foreground colors
<code>A_STANDOUT</code>	Standout mode
<code>A_UNDERLINE</code>	Underline mode
<code>A_HORIZONTAL</code>	Horizontal highlight
<code>A_LEFT</code>	Left highlight
<code>A_LOW</code>	Low highlight
<code>A_RIGHT</code>	Right highlight
<code>A_TOP</code>	Top highlight
<code>A_VERTICAL</code>	Vertical highlight
<code>A_CHARTEXT</code>	Bit-mask to extract a character

Several constants are available to extract corresponding attributes returned by some methods.

Bit-mask	Signification
A_ATTRIBUTES	Bit-mask to extract attributes
A_CHARTEXT	Bit-mask to extract a character
A_COLOR	Bit-mask to extract color-pair field information

Keys are referred to by integer constants with names starting with `KEY_`. The exact keycaps available are system dependent.

Key constant	Key
KEY_MIN	Minimum key value
KEY_BREAK	Break key (unreliable)
KEY_DOWN	Down-arrow
KEY_UP	Up-arrow
KEY_LEFT	Left-arrow
KEY_RIGHT	Right-arrow
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	Backspace (unreliable)
KEY_F0	Function keys. Up to 64 function keys are supported.
KEY_Fn	Value of function key <i>n</i>
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send (unreliable)
KEY_SRESET	Soft (partial) reset (unreliable)
KEY_RESET	Reset or hard reset (unreliable)
KEY_PRINT	Print
KEY_LL	Home down or bottom (lower left)
KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad
KEY_BTAB	Back tab
KEY_BEG	Beg (beginning)
KEY_CANCEL	Cancel
KEY_CLOSE	<i>Close</i>
KEY_COMMAND	Cmd (command)
KEY_COPY	<i>Copy</i>

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Key constant	Key
KEY_CREATE	Create
KEY_END	End
KEY_EXIT	<i>Exit</i>
KEY_FIND	Find
KEY_HELP	Help
KEY_MARK	Mark
KEY_MESSAGE	Message
KEY_MOVE	Move
KEY_NEXT	Next
KEY_OPEN	Open
KEY_OPTIONS	Options
KEY_PREVIOUS	Prev (previous)
KEY_REDO	<i>Redo</i>
KEY_REFERENCE	Ref (reference)
KEY_REFRESH	Refresh
KEY_REPLACE	Replace
KEY_RESTART	Restart
KEY_RESUME	Resume
KEY_SAVE	<i>Save</i>
KEY_SBEG	Shifted Beg (beginning)
KEY_SCANCEL	Shifted Cancel
KEY_SCOMMAND	Shifted Command
KEY_SCOPY	Shifted Copy
KEY_SCREATE	Shifted Create
KEY_SDC	Shifted Delete char
KEY_SDL	Shifted Delete line
KEY_SELECT	Select
KEY_SEND	Shifted End
KEY_SEOL	Shifted Clear line
KEY_SEXIT	Shifted Exit
KEY_SFIND	Shifted Find
KEY_SHELP	Shifted Help
KEY_SHOME	Shifted Home
KEY_SIC	Shifted Input
KEY_SLEFT	Shifted Left arrow
KEY_SMESSAGE	Shifted Message
KEY_SMOVE	Shifted Move
KEY_SNEXT	Shifted Next
KEY_SOPTIONS	Shifted Options
KEY_SPREVIOUS	Shifted Prev
KEY_SPRINT	Shifted Print
KEY_SREDO	Shifted Redo
KEY_SREPLACE	Shifted Replace
KEY_SRIGHT	Shifted Right arrow
KEY_SRSUME	Shifted Resume
KEY_SSAVE	Shifted Save
KEY_SSUSPEND	Shifted Suspend
KEY_SUNDO	Shifted Undo
KEY_SUSPEND	Suspend

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Key constant	Key
KEY_UNDO	<i>Undo</i>
KEY_MOUSE	Mouse event has occurred
KEY_RESIZE	Terminal resize event
KEY_MAX	Maximum key value

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (KEY_F1, KEY_F2, KEY_F3, KEY_F4) available, and the arrow keys mapped to KEY_UP, KEY_DOWN, KEY_LEFT and KEY_RIGHT in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard :

Keycap	Constante
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

Note : These are available only after `initscr()` has been called.

ACS code	Signification
ACS_BBSS	alternate name for upper right corner
ACS_BLOCK	solid square block
ACS_BOARD	board of squares
ACS_BSBS	alternate name for horizontal line
ACS_BSSB	alternate name for upper left corner
ACS_BSSS	alternate name for top tee
ACS_BTEE	bottom tee
ACS_BULLET	bullet
ACS_CKBOARD	checker board (stipple)
ACS_DARROW	arrow pointing down
ACS_DEGREE	degree symbol
ACS_DIAMOND	diamond
ACS_GEQUAL	greater-than-or-equal-to
ACS_HLINE	horizontal line
ACS_LANTERN	lantern symbol
ACS_LARROW	left arrow
ACS_LEQUAL	less-than-or-equal-to
ACS_LLCORNER	lower left-hand corner
ACS_LRCORNER	lower right-hand corner
ACS_LTEE	left tee
ACS_NEQUAL	not-equal sign
ACS_PI	letter pi

Suite sur la page suivante

Tableau 2 – suite de la page précédente

ACS code	Signification
ACS_PLMINUS	plus-or-minus sign
ACS_PLUS	big plus sign
ACS_RARROW	right arrow
ACS_RTEE	right tee
ACS_S1	scan line 1
ACS_S3	scan line 3
ACS_S7	scan line 7
ACS_S9	scan line 9
ACS_SBBS	alternate name for lower right corner
ACS_SBSB	alternate name for vertical line
ACS_SBSS	alternate name for right tee
ACS_SSBB	alternate name for lower left corner
ACS_SSBS	alternate name for bottom tee
ACS_SSSB	alternate name for left tee
ACS_SSSS	alternate name for crossover or big plus
ACS_STERLING	pound sterling
ACS_TTEE	top tee
ACS_UARROW	up arrow
ACS_ULCORNER	upper left corner
ACS_URCORNER	upper right corner
ACS_VLINE	vertical line

The following table lists the predefined colors :

Constante	Color
COLOR_BLACK	Black
COLOR_BLUE	Blue
COLOR_CYAN	Cyan (light greenish blue)
COLOR_GREEN	Green
COLOR_MAGENTA	Magenta (purplish red)
COLOR_RED	Red
COLOR_WHITE	White
COLOR_YELLOW	Yellow

16.11 `curses.textpad` — Text input widget for curses programs

The `curses.textpad` module provides a `Textbox` class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module `curses.textpad` defines the following function :

`curses.textpad.rectangle` (*win*, *uly*, *ulx*, *lry*, *lrx*)

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

16.11.1 Textbox objects

You can instantiate a *Textbox* object as follows :

class `curses.textpad.Textbox` (*win*)

Return a textbox widget object. The *win* argument should be a curses *window* object in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates (0, 0). The instance's *stripspaces* flag is initially on.

Textbox objects have the following methods :

edit ([*validator*])

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If *validator* is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* attribute.

do_command (*ch*)

Process a single command keystroke. Here are the supported special keystrokes :

Keystroke	Action
Control-A	Go to left edge of window.
Control-B	Cursor left, wrapping to previous line if appropriate.
Control-D	Delete character under cursor.
Control-E	Go to right edge (stripspaces off) or end of line (stripspaces on).
Control-F	Cursor right, wrapping to next line when appropriate.
Control-G	Terminate, returning the window contents.
Control-H	Delete character backward.
Control-J	Terminate if the window is 1 line, otherwise insert newline.
Control-K	If line is blank, delete it, otherwise clear to end of line.
Control-L	Refresh screen.
Control-N	Cursor down; move down one line.
Control-O	Insert a blank line at cursor location.
Control-P	Cursor up; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible :

Constante	Keystroke
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

gather ()

Return the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* member.

stripspaces

This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

16.12 `curses.ascii` — Utilities for ASCII characters

The `curses.ascii` module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows :

Nom	Signification
NUL	
SOH	Start of heading, console interrupt
STX	Start of text
ETX	End of text
EOT	End of transmission
ENQ	Enquiry, goes with ACK flow control
ACK	Acknowledgement
BEL	Bell
BS	Backspace
TAB	Tab
HT	Alias for TAB : « Horizontal tab »
LF	Line feed
NL	Alias for LF : « New line »
VT	Vertical tab
FF	Form feed
CR	Retour chariot
SO	Shift-out, begin alternate character set
SI	Shift-in, resume default character set
DLE	Data-link escape
DC1	XON, for flow control
DC2	Device control 2, block-mode flow control
DC3	XOFF, for flow control
DC4	Device control 4
NAK	Negative acknowledgement
SYN	Synchronous idle
ETB	End transmission block
CAN	Cancel
EM	End of medium
SUB	Substitute
ESC	Escape
FS	Séparateur de fichiers
GS	Séparateur de groupe
RS	Record separator, block-mode terminator
US	Unit separator
SP	Space
DEL	Delete

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library :

`curses.ascii.isalnum(c)`

Checks for an ASCII alphanumeric character; it is equivalent to `isalpha(c)` or `isdigit(c)`.

`curses.ascii.isalpha(c)`

Checks for an ASCII alphabetic character; it is equivalent to `isupper(c)` or `islower(c)`.

`curses.ascii.isascii(c)`

Checks for a character value that fits in the 7-bit ASCII set.

`curses.ascii.isblank(c)`

Checks for an ASCII whitespace character; space or horizontal tab.

`curses.ascii.iscntrl(c)`

Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

`curses.ascii.isdigit(c)`

Checks for an ASCII decimal digit, '0' through '9'. This is equivalent to `c in string.digits`.

`curses.ascii.isgraph(c)`

Checks for ASCII any printable character except space.

`curses.ascii.islower(c)`

Checks for an ASCII lower-case character.

`curses.ascii.isprint(c)`

Checks for any ASCII printable character including space.

`curses.ascii.ispunct(c)`

Checks for any printable ASCII character which is not a space or an alphanumeric character.

`curses.ascii.isspace(c)`

Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

`curses.ascii.isupper(c)`

Checks for an ASCII uppercase letter.

`curses.ascii.isxdigit(c)`

Checks for an ASCII hexadecimal digit. This is equivalent to `c in string.hexdigits`.

`curses.ascii.isctrl(c)`

Checks for an ASCII control character (ordinal values 0 to 31).

`curses.ascii.ismeta(c)`

Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or single-character strings; when the argument is a string, it is first converted using the built-in function `ord()`.

Note that all these functions check ordinal bit values derived from the character of the string you pass in; they do not actually know anything about the host machine's character encoding.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

`curses.ascii.ascii(c)`

Return the ASCII value corresponding to the low 7 bits of `c`.

`curses.ascii.ctrl(c)`

Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

`curses.ascii.alt(c)`

Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

`curses.ascii.unctrl(c)`

Return a string representation of the ASCII character *c*. If *c* is printable, this string is the character itself. If the character is a control character (0x00–0x1f) the string consists of a caret ('^') followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is '^?'. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and '!' prepended to the result.

`curses.ascii.controlnames`

A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic *SP* for the space character.

16.13 `curses.panel` — A panel stack extension for `curses`

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

16.13.1 Fonctions

The module `curses.panel` defines the following functions :

`curses.panel.bottom_panel()`

Returns the bottom panel in the panel stack.

`curses.panel.new_panel(win)`

Returns a panel object, associating it with the given window *win*. Be aware that you need to keep the returned panel object referenced explicitly. If you don't, the panel object is garbage collected and removed from the panel stack.

`curses.panel.top_panel()`

Returns the top panel in the panel stack.

`curses.panel.update_panels()`

Updates the virtual screen after changes in the panel stack. This does not call `curses.doupdate()`, so you'll have to do this yourself.

16.13.2 Panel Objects

Panel objects, as returned by `new_panel()` above, are windows with a stacking order. There's always a window associated with a panel which determines the content, while the panel methods are responsible for the window's depth in the panel stack.

Panel objects have the following methods :

`Panel.above()`

Returns the panel above the current panel.

`Panel.below()`

Returns the panel below the current panel.

`Panel.bottom()`

Push the panel to the bottom of the stack.

`Panel.hidden()`

Returns `True` if the panel is hidden (not visible), `False` otherwise.

`Panel.hide()`

Hide the panel. This does not delete the object, it just makes the window on screen invisible.

`Panel.move(y, x)`

Move the panel to the screen coordinates `(y, x)`.

`Panel.replace(win)`

Change the window associated with the panel to the window `win`.

`Panel.set_userptr(obj)`

Set the panel's user pointer to `obj`. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

`Panel.show()`

Display the panel (which might have been hidden).

`Panel.top()`

Push panel to the top of the stack.

`Panel.userptr()`

Returns the user pointer for the panel. This might be any Python object.

`Panel.window()`

Returns the window object associated with the panel.

16.14 platform — Access to underlying platform's identifying data

Code source : [Lib/platform.py](#)

Note : Specific platforms listed alphabetically, with Linux included in the Unix section.

16.14.1 Cross Platform

`platform.architecture(executable=sys.executable, bits="", linkage="")`

Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple `(bits, linkage)` which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If `bits` is given as `''`, the `sizeof(pointer)` (or `sizeof(long)` on Python version < 1.5.2) is used as indicator for the supported pointer size.

The function relies on the system's `file` command to do the actual work. This is available on most if not all Unix platforms and some non-Unix platforms and then only if the executable points to the Python interpreter. Reasonable defaults are used when the above needs are not met.

Note : On Mac OS X (and perhaps other platforms), executable files may be universal files containing multiple architectures.

To get at the « 64-bitness » of the current interpreter, it is more reliable to query the `sys.maxsize` attribute :

```
is_64bits = sys.maxsize > 2**32
```

`platform.machine()`
Returns the machine type, e.g. 'i386'. An empty string is returned if the value cannot be determined.

`platform.node()`
Returns the computer's network name (may not be fully qualified!). An empty string is returned if the value cannot be determined.

`platform.platform(aliased=0, terse=0)`
Returns a single string identifying the underlying platform with as much useful information as possible. The output is intended to be *human readable* rather than machine parseable. It may look different on different platforms and this is intended. If *aliased* is true, the function will use aliases for various platforms that report system names which differ from their common names, for example SunOS will be reported as Solaris. The `system_alias()` function is used to implement this. Setting *terse* to true causes the function to return only the absolute minimum information needed to identify the platform.

`platform.processor()`
Returns the (real) processor name, e.g. 'amd64'. An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for `machine()`. NetBSD does this.

`platform.python_build()`
Returns a tuple (buildno, builddate) stating the Python build number and date as strings.

`platform.python_compiler()`
Returns a string identifying the compiler used for compiling Python.

`platform.python_branch()`
Returns a string identifying the Python implementation SCM branch.

`platform.python_implementation()`
Returns a string identifying the Python implementation. Possible return values are : "CPython", "IronPython", "Jython", "PyPy".

`platform.python_revision()`
Returns a string identifying the Python implementation SCM revision.

`platform.python_version()`
Returns the Python version as string 'major.minor.patchlevel'. Note that unlike the Python `sys.version`, the returned value will always include the patchlevel (it defaults to 0).

`platform.python_version_tuple()`
Returns the Python version as tuple (major, minor, patchlevel) of strings. Note that unlike the Python `sys.version`, the returned value will always include the patchlevel (it defaults to '0').

`platform.release()`
Returns the system's release, e.g. '2.2.0' or 'NT'. An empty string is returned if the value cannot be determined.

`platform.system()`
Returns the system/OS name, e.g. 'Linux', 'Windows', or 'Java'. An empty string is returned if the value cannot be determined.

`platform.system_alias(system, release, version)`
Returns (system, release, version) aliased to common marketing names used for some systems. It also does some reordering of the information in some cases where it would otherwise cause confusion.

`platform.version()`

Returns the system's release version, e.g. `'#3 on degas'`. An empty string is returned if the value cannot be determined.

`platform.uname()`

Fairly portable `uname` interface. Returns a `namedtuple()` containing six attributes : `system`, `node`, `release`, `version`, `machine`, and `processor`.

Note that this adds a sixth attribute (`processor`) not present in the `os.uname()` result. Also, the attribute names are different for the first two attributes; `os.uname()` names them `sysname` and `nodename`.

Entries which cannot be determined are set to `' '`.

Modifié dans la version 3.3 : Le type renvoyé passe d'un `tuple` à un `namedtuple`.

16.14.2 Java Platform

`platform.java_ver(release="", vendor="", vminfo=(",",), osinfo=(",",))`

Version interface for Jython.

Returns a tuple `(release, vendor, vminfo, osinfo)` with `vminfo` being a tuple `(vm_name, vm_release, vm_vendor)` and `osinfo` being a tuple `(os_name, os_version, os_arch)`. Values which cannot be determined are set to the defaults given as parameters (which all default to `' '`).

16.14.3 Windows Platform

`platform.win32_ver(release="", version="", csd="", ptype="")`

Get additional version information from the Windows Registry and return a tuple `(release, version, csd, ptype)` referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor).

As a hint : `ptype` is `'Uniprocessor Free'` on single processor NT machines and `'Multiprocessor Free'` on multi processor machines. The *"Free"* refers to the OS version being free of debugging code. It could also state *"Checked"* which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

Note : This function works best with Mark Hammond's `win32all` package installed, but also on Python 2.3 and later (support for this was added in Python 2.6). It obviously only runs on Win32 compatible platforms.

Win95/98 specific

`platform.popen(cmd, mode='r', bufsize=-1)`

Portable `popen()` interface. Find a working `popen` implementation preferring `win32pipe.popen()`. On Windows NT, `win32pipe.popen()` should work; on Windows 9x it hangs due to bugs in the MS C library.

Obsolète depuis la version 3.3 : This function is obsolete. Use the `subprocess` module. Check especially the *Remplacer les fonctions plus anciennes par le module subprocess* section.

16.14.4 Mac OS Platform

`platform.mac_ver` (*release*=", *versioninfo*=(", ", "), *machine*=")

Get Mac OS version information and return it as tuple (*release*, *versioninfo*, *machine*) with *versioninfo* being a tuple (*version*, *dev_stage*, *non_release_version*).

Entries which cannot be determined are set to ' '. All tuple entries are strings.

16.14.5 Unix Platforms

`platform.dist` (*distname*=", *version*=", *id*=", *supported_dists*=(*'SuSE'*, *'debian'*, *'redhat'*, *'mandrake'*, ...))

This is another name for `linux_distribution()`.

Deprecated since version 3.5, will be removed in version 3.8 : See alternative like the `distro` package.

`platform.linux_distribution` (*distname*=", *version*=", *id*=", *supported_dists*=(*'SuSE'*, *'debian'*, *'redhat'*, *'mandrake'*, ...), *full_distribution_name*=1)

Tries to determine the name of the Linux OS distribution name.

supported_dists may be given to define the set of Linux distributions to look for. It defaults to a list of currently supported Linux distributions identified by their release file name.

If *full_distribution_name* is true (default), the full distribution read from the OS is returned. Otherwise the short name taken from *supported_dists* is used.

Returns a tuple (*distname*, *version*, *id*) which defaults to the args given as parameters. *id* is the item in parentheses after the version number. It is usually the version codename.

Deprecated since version 3.5, will be removed in version 3.8 : See alternative like the `distro` package.

`platform.libc_ver` (*executable*=*sys.executable*, *lib*=", *version*=", *chunksize*=16384)

Tries to determine the libc version against which the file *executable* (defaults to the Python interpreter) is linked.

Returns a tuple of strings (*lib*, *version*) which default to the given parameters in case the lookup fails.

Note that this function has intimate knowledge of how different libc versions add symbols to the executable is probably only usable for executables compiled using `gcc`.

The file is read and scanned in chunks of *chunksize* bytes.

16.15 `errno` — Symboles du système *errno* standard

Ce module met à disposition des symboles du système standard `errno`. La valeur de chaque symbole est la valeur entière correspondante. Les noms et les descriptions sont empruntés à `linux/include/errno.h`, qui devrait être assez exhaustif.

`errno.errorcode`

Dictionnaire associant la valeur *errno* au nom de chaîne dans le système sous-jacent. Par exemple, `errno.errorcode[errno.EPERM]` correspond à `'EPERM'`.

Pour traduire un code d'erreur en message d'erreur, utilisez `os.strerror()`.

De la liste suivante, les symboles qui ne sont pas utilisés dans la plateforme actuelle ne sont pas définis par le module. La liste spécifique des symboles définis est disponible comme `errno.errorcode.keys()`. Les symboles disponibles font partie de cette liste :

`errno.EPERM`

Opération interdite

`errno.ENOENT`

Fichier ou répertoire inexistant

`errno.ESRCH`
Processus inexistant

`errno.EINTR`
Appel système interrompu
Voir aussi :
Cette erreur est associée à l'exception *InterruptedError*.

`errno.EIO`
Erreur d'entrée-sortie

`errno.ENXIO`
Dispositif ou adresse inexistant

`errno.E2BIG`
Liste d'arguments trop longue

`errno.ENOEXEC`
Erreur de format d'exécution

`errno.EBADF`
Mauvais descripteur de fichier

`errno.ECHILD`
Pas de processus fils

`errno.EAGAIN`
Ressource temporairement indisponible (réessayez)

`errno.ENOMEM`
Mémoire insuffisante

`errno.EACCES`
Autorisation refusée

`errno.EFAULT`
Mauvaise adresse

`errno.ENOTBLK`
Dispositif de bloc requis

`errno.EBUSY`
Dispositif ou ressource occupé

`errno.EEXIST`
Fichier déjà existant

`errno.EXDEV`
Lien inapproprié

`errno.ENODEV`
Dispositif inexistant

`errno.ENOTDIR`
Pas un répertoire

`errno.EISDIR`
Est un répertoire

`errno.EINVAL`
Argument invalide

`errno.ENFILE`
Plus de descripteur de fichier disponible

`errno.EMFILE`
Trop de fichiers ouverts

`errno.ENOTTY`
Opération de contrôle d'entrée-sortie invalide

`errno.ETXTBSY`
Fichier texte occupé

`errno.EFBIG`
Fichier trop grand

`errno.ENOSPC`
Plus de place sur le dispositif

`errno.ESPIPE`
Recherche invalide

`errno.EROFS`
Système de fichiers en lecture seule

`errno.EMLINK`
Trop de liens symboliques

`errno.EPIPE`
Tube brisé

`errno.EDOM`
Argument mathématique hors du domaine de définition de la fonction

`errno.ERANGE`
Résultat mathématique non représentable

`errno.EDEADLK`
Un interblocage se produirait sur cette ressource

`errno.ENAMETOOLONG`
Nom de fichier trop long

`errno.ENOLCK`
Plus de verrou de fichier disponible

`errno.ENOSYS`
Fonction non implémentée

`errno.ENOTEMPTY`
Dossier non vide

`errno.ELOOP`
Trop de liens symboliques trouvés

`errno.EWOULDBLOCK`
L'opération bloquerait

`errno.ENOMSG`
Pas de message du type voulu

`errno.EIDRM`
Identifiant supprimé

`errno.ECHRNG`
Le numéro de canal est hors des limites

`errno.EL2NSYNC`
Le niveau 2 n'est pas synchronisé

`errno.EL3HLT`
Niveau 3 stoppé

`errno.EL3RST`
Niveau 3 réinitialisé

`errno.ELNRNG`
Le numéro du lien est hors des limites

`errno.EUNATCH`
Le pilote de protocole n'est pas attaché

`errno.ENOCSI`
Pas de structure *CSI* disponible

`errno.EL2HLT`
Niveau 2 stoppé

`errno.EBADE`
Échange invalide

`errno.EBADR`
Descripteur de requête invalide

`errno.EXFULL`
Échange complet

`errno.ENOANO`
Pas de *anode*

`errno.EBADRQC`
Code de requête invalide

`errno.EBADSLT`
Slot invalide

`errno.EDEADLOCK`
Interblocage lors du verrouillage de fichier

`errno.EBFONT`
Mauvais format de fichier de police

`errno.ENOSTR`
Le périphérique n'est pas un flux

`errno.ENODATA`
Pas de donnée disponible

`errno.ETIME`
Délai maximal atteint

`errno.ENOSR`
Pas assez de ressources de type flux

`errno.ENONET`
Machine hors réseau

`errno.ENOPKG`
Paquet non installé

`errno.EREMOTE`
L'objet est distant

`errno.ENOLINK`
Lien coupé

`errno.EADV`
Erreur d'annonce

`errno.ESRMNT`
Erreur *Srmount*

`errno.ECOMM`
Erreur de communication lors de l'envoi

`errno.EPROTO`
Erreur de protocole

`errno.EMULTIHOP`
Transfert à sauts multiples essayé

`errno.EDOTDOT`
erreur spécifique *RFS*

`errno.EBADMSG`
Pas un message de données

`errno.EOVERFLOW`
Valeur trop grande pour être stockée dans ce type de donnée

`errno.ENOTUNIQ`
Nom non-unique dans le réseau

`errno.EBADFD`
Descripteur de fichier en mauvais état

`errno.EREMCHG`
Adresse distante changée

`errno.ELIBACC`
Accès impossible à une bibliothèque partagée nécessaire

`errno.ELIBBAD`
Accès à une bibliothèque partagée corrompue

`errno.ELIBSCN`
Section *.lib* de *a.out* corrompue

`errno.ELIBMAX`
Tentative de liaison entre trop de bibliothèques partagées

`errno.ELIBEXEC`
Impossible d'exécuter directement une bibliothèque partagée

`errno.EILSEQ`
Séquence de *bytes* illégale

`errno.ERESTART`
Appel système interrompu qui devrait être relancé

`errno.ESTRPIPE`
Erreur d'enchaînement de flux

`errno.EUSERS`
Trop d'utilisateurs

`errno.ENOTSOCK`
Opération d'interface de connexion alors que ce n'est pas une interface de connexion

`errno.EDESTADDRREQ`
Adresse de destination obligatoire

`errno.EMSGSIZE`
Message trop long

`errno.EPROTOTYPE`
Mauvais type de protocole pour ce connecteur

`errno.ENOPROTOOPT`
Protocole pas disponible

`errno.EPROTONOSUPPORT`
Protocole non géré

`errno.ESOCKTNOSUPPORT`
Type de connecteur non géré

`errno.EOPNOTSUPP`
Opération non gérée par cette fin de lien

`errno.EPFNOSUPPORT`
Famille de protocole non gérée

`errno.EAFNOSUPPORT`
Famille d'adresses non gérée par ce protocole

`errno.EADDRINUSE`
Adresse déjà utilisée

`errno.EADDRNOTAVAIL`
Impossible d'assigner l'adresse demandée

`errno.ENETDOWN`
Le réseau est désactivé

`errno.ENETUNREACH`
Réseau inaccessible

`errno.ENETRESET`
Connexion annulée par le réseau

`errno.ECONNABORTED`
Connexion abandonnée

`errno.ECONNRESET`
Connexion réinitialisée

`errno.ENOBUFS`
Plus d'espace tampon disponible

`errno.EISCONN`
L'interface de connexion est déjà connectée

`errno.ENOTCONN`
L'interface de connexion n'est pas connectée

`errno.ESHUTDOWN`
Impossible d'envoyer après l'arrêt du point final du transport

`errno.ETOOMANYREFS`
Trop de descripteurs : impossible d'effectuer la liaison

`errno.ETIMEDOUT`
Délai maximal de connexion écoulé

`errno.ECONNREFUSED`
Connexion refusée

`errno.EHOSTDOWN`
Hôte éteint

`errno.EHOSTUNREACH`
Pas de route vers l'hôte

`errno.EALREADY`
Connexion déjà en cours

`errno.EINPROGRESS`
Opération en cours

`errno.ESTALE`
Descripteur de fichier NFS corrompu

`errno.EUCLEAN`
La structure a besoin d'être nettoyée

`errno.ENOTNAM`
N'est pas un fichier nommé du type *XENIX*

`errno.ENAVAIL`
Pas de sémaphore *XENIX* disponible

`errno.EISNAM`
Est un fichier nommé

`errno.EREMOTEIO`
Erreur d'entrées-sorties distante

`errno.EDQUOT`
Quota dépassé

16.16 `ctypes` — Bibliothèque Python d'appels à des fonctions externes

`ctypes` est une bibliothèque d'appel à des fonctions externes en python. Elle fournit des types de données compatibles avec le langage C et permet d'appeler des fonctions depuis des DLL ou des bibliothèques partagées, rendant ainsi possible l'interfaçage de ces bibliothèques avec du pur code Python.

16.16.1 Didacticiel de *ctypes*

Remarque : Les exemples de code de ce didacticiel utilisent *doctest* pour s'assurer de leur propre bon fonctionnement. Vu que certains de ces exemples ont un comportement différent en Linux, Windows ou Mac OS X, ils contiennent des directives *doctest* dans les commentaires.

Remarque : Le type *c_int* du module apparaît dans certains de ces exemples. Sur les plates-formes où `sizeof(long) == sizeof(int)`, ce type est un alias de *c_long*. Ne soyez donc pas surpris si *c_long* s'affiche là où vous vous attendiez à *c_int* — il s'agit bien du même type.

Chargement des DLL

ctypes fournit l'objet *cdll* pour charger des bibliothèques à liens dynamiques (et les objets *windll* et *oledll* en Windows).

Une bibliothèque se charge en y accédant comme un attribut de ces objets. *cdll* charge les bibliothèques qui exportent des fonctions utilisant la convention d'appel standard *cdecl*, alors que les bibliothèques qui se chargent avec *windll* utilisent la convention d'appel *stdcall*. *oledll* utilise elle aussi la convention *stdcall* et suppose que les fonctions renvoient un code d'erreur HRESULT de Windows. Ce code d'erreur est utilisé pour lever automatiquement une *OSError* quand l'appel de la fonction échoue.

Modifié dans la version 3.3 : En Windows, les erreurs levaient auparavant une *WindowsError*, qui est maintenant un alias de *OSError*.

Voici quelques exemples Windows. *msvcrt* est la bibliothèque standard C de Microsoft qui contient la plupart des fonctions standards C. Elle suit la convention d'appel *cdecl* :

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows ajoute le suffixe habituel *.dll* automatiquement.

Note : Accéder à la bibliothèque standard C par *cdll.msvcrt* utilise une version obsolète de la bibliothèque qui peut avoir des problèmes de compatibilité avec celle que Python utilise. Si possible, mieux vaut utiliser la fonctionnalité native de Python, ou bien importer et utiliser le module *msvcrt*.

Pour charger une bibliothèque en Linux, il faut passer le nom du fichier *avec* son extension. Il est donc impossible de charger une bibliothèque en accédant à un attribut. Il faut utiliser la méthode *LoadLibrary()* des chargeurs de DLL, ou bien charger la bibliothèque en créant une instance de *CDLL* en appelant son constructeur :

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```


Accès aux fonctions des DLL chargées

Les fonctions sont alors des attributs des objets DLL :

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

Les DLL des systèmes *win32* comme *kernel32* et *user32* exportent souvent une version ANSI et une version UNICODE d'une fonction. La version UNICODE est exportée avec un W à la fin, et la version ANSI avec un A. La fonction *win32* *GetModuleHandle*, qui renvoie un *gestionnaire de module* à partir de son nom, a le prototype C suivant (c'est une macro qui décide d'exporter l'une ou l'autre à travers *GetModuleHandle*, selon qu'UNICODE est définie ou non) :

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll n'en choisit pas une par magie, il faut accéder à la bonne en écrivant explicitement *GetModuleHandleA* ou *GetModuleHandleW* et en les appelant ensuite avec des objets octets ou avec des chaînes de caractères, respectivement.

Les DLL exportent parfois des fonctions dont les noms ne sont pas des identifiants Python valides, comme "??2@YAPAXI@Z". Dans ce cas, il faut utiliser *getattr()* pour accéder à la fonction :

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

Sous Windows, certaines DLL exportent des fonctions à travers un indice plutôt qu'à travers un nom. On accède à une fonction en indiquant l'objet DLL avec son index :

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

Appel de fonctions

Ces fonctions s'appellent comme n'importe quel callable Python. Cet exemple utilise la fonction `time()`, qui renvoie le temps en secondes du système depuis l'*epoch* Unix, et la fonction `GetModuleHandleA()`, qui renvoie un gestionnaire de module *win32*.

This example calls both functions with a NULL pointer (`None` should be used as the NULL pointer) :

```
>>> print(libc.time(None))
1150640792
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

Note : `ctypes` may raise a `ValueError` after calling the function, if it detects that an invalid number of arguments were passed. This behavior should not be relied upon. It is deprecated in 3.6.2, and will be removed in 3.7.

Une `ValueError` est levée quand on appelle une fonction `stdcall` avec la convention d'appel `cdecl` et vice-versa :

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

Pour déterminer la convention d'appel, il faut consulter l'en-tête C ou la documentation de la fonction à appeler.

En Windows, `ctypes` tire profit de la gestion structurée des exceptions (*structured exception handling*) *win32* pour empêcher les plantages dus à des interruptions, afin de préserver la protection globale (*general protection faults*) du système, lorsque des fonctions sont appelées avec un nombre incorrect d'arguments :

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

Cependant, il y a suffisamment de façons de faire planter Python avec `ctypes`, donc il faut être prudent dans tous les cas. Le module `faulthandler` est pratique pour déboguer les plantages (p. ex. dus à des erreurs de segmentation produites par des appels erronés à la bibliothèque C).

`None`, les entiers, les objets octets et les chaînes de caractères (Unicode) sont les seuls types natifs de Python qui peuvent être directement passés en paramètres de ces appels de fonctions. `None` est passé comme le pointeur C `NULL`, les objets octets et les chaînes de caractères sont passés comme un pointeur sur le bloc mémoire qui contient la donnée (`char *` ou `wchar_t *`). Les entiers Python sont passés comme le type C `int` par défaut de la plate-forme, un masque étant appliqué pour qu'ils tiennent dans le type C.

Avant de poursuivre sur l'appel de fonctions avec d'autres types de paramètres, apprenons-en un peu plus sur les types de données de `ctypes`.

Types de données fondamentaux

`ctypes` définit plusieurs types de donnée de base compatibles avec le C :

Types de <code>ctypes</code>	Type C	Type Python
<code>c_bool</code>	<code>_Bool</code>	<code>bool</code> (1)
<code>c_char</code>	<code>char</code>	objet octet (<i>bytes</i>) de 1 caractère
<code>c_wchar</code>	<code>wchar_t</code>	chaîne de caractères (<i>string</i>) de longueur 1
<code>c_byte</code>	<code>char</code>	<code>int</code>
<code>c_ubyte</code>	<code>unsigned char</code>	<code>int</code>
<code>c_short</code>	<code>short</code>	<code>int</code>
<code>c_ushort</code>	<code>unsigned short</code>	<code>int</code>
<code>c_int</code>	<code>int</code>	<code>int</code>
<code>c_uint</code>	<code>unsigned int</code>	<code>int</code>
<code>c_long</code>	<code>long</code>	<code>int</code>
<code>c_ulong</code>	<code>unsigned long</code>	<code>int</code>
<code>c_longlong</code>	<code>__int64</code> ou <code>long long</code>	<code>int</code>
<code>c_ulonglong</code>	<code>unsigned __int64</code> ou <code>unsigned long long</code>	<code>int</code>
<code>c_size_t</code>	<code>size_t</code>	<code>int</code>
<code>c_ssize_t</code>	<code>ssize_t</code> ou <code>Py_ssize_t</code>	<code>int</code>
<code>c_float</code>	<code>float</code>	<code>float</code>
<code>c_double</code>	<code>double</code>	<code>float</code>
<code>c_longdouble</code>	<code>long double</code>	<code>float</code>
<code>c_char_p</code>	<code>char *</code> (terminé par NUL)	objet octet (<i>bytes</i>) ou <code>None</code>
<code>c_wchar_p</code>	<code>wchar_t *</code> (terminé par NUL)	chaîne de caractères (<i>string</i>) ou <code>None</code>
<code>c_void_p</code>	<code>void *</code>	<code>int</code> ou <code>None</code>

(1) Le constructeur accepte n'importe quel objet convertible en booléen.

Il est possible de créer chacun de ces types en les appelant avec une valeur d'initialisation du bon type et avec une valeur cohérente :

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

Ces types étant des muables, leur valeur peut aussi être modifiée après coup :

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

Affecter une nouvelle valeur à une instance de type pointeur — `c_char_p`, `c_wchar_p` et `c_void_p` — change la zone mémoire sur laquelle elle pointe, et non le contenu de ce bloc mémoire (c'est logique parce que les objets octets sont immuables en Python) :

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)                # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                  # first object is unchanged
Hello, World
>>>
```

Cependant, prenez garde à ne pas en passer à des fonctions qui prennent en paramètre des pointeurs sur de la mémoire modifiable. S'il vous faut de la mémoire modifiable, `ctypes` fournit la fonction `create_string_buffer()` qui en crée de plusieurs façons. L'attribut `raw` permet d'accéder à (ou de modifier) un bloc mémoire ; l'attribut `value` permet d'y accéder comme à une chaîne de caractères terminée par NUL :

```
>>> from ctypes import *
>>> p = create_string_buffer(3)                # create a 3 byte buffer, initialized to
↳NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")        # create a buffer containing a NUL
↳terminated string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10)    # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

La fonction `create_string_buffer()` remplace les fonctions `c_buffer()` (qui en reste un alias) et `c_string()` des versions antérieures de `ctypes`. La fonction `create_unicode_buffer()` crée un bloc mémoire modifiable contenant des caractères Unicode du type C `wchar_t`.

Appel de fonctions, suite

`printf` utilise la vraie sortie standard, et non `sys.stdout` ; les exemples suivants ne fonctionnent donc que dans une invite de commande et non depuis `IDLE` or `PythonWin` :

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert parameter 2
>>>
```

Comme mentionné plus haut, tous les types Python (les entiers, les chaînes de caractères et les objets octet exceptés) doivent être encapsulés dans leur type `ctypes` correspondant pour pouvoir être convertis dans le type C requis :

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

Appel de fonctions avec des types de données personnalisés

Il est possible de personnaliser la conversion des arguments effectuée par `ctypes` pour permettre de passer en argument des instances de vos propres classes. `ctypes` recherche un attribut `_as_parameter_` et le prend comme argument à la fonction. Bien entendu, cet attribut doit être un entier, une chaîne de caractères ou des octets :

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

Si vous ne souhaitez pas stocker les données de l'instance dans la variable `_as_parameter_` de l'instance, vous pouvez toujours définir une *propriété* qui rend cet attribut disponible sur demande.

Définition du type des arguments nécessaires (prototypes de fonction)

Il est possible de définir le type des arguments demandés par une fonction exportée depuis une DLL en définissant son attribut `argtypes`.

`argtypes` doit être une séquence de types de données C (la fonction `printf` n'est probablement pas le meilleur exemple pour l'illustrer, car elle accepte un nombre variable d'arguments de types eux aussi variables, selon la chaîne de formatage ; cela dit, elle se révèle pratique pour tester cette fonctionnalité) :

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

Définir un format empêche de passer des arguments de type incompatible (comme le fait le prototype d'une fonction C) et tente de convertir les arguments en des types valides :

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

Pour appeler une fonction avec votre propre classe définie dans la séquence `argtypes`, il est nécessaire d'implémenter une méthode de classe `from_param()`. La méthode de classe `from_param()` récupère l'objet Python passé à la fonction et doit faire une vérification de type ou tout ce qui est nécessaire pour s'assurer que l'objet est valide, puis renvoie l'objet lui-même, son attribut `_as_parameter_`, ou tout ce que vous voulez passer comme argument fonction C dans ce cas. Encore une fois, il convient que le résultat soit un entier, une chaîne, des octets, une instance `ctypes` ou un objet avec un attribut `_as_parameter_`.

Types de sortie

Le module suppose que toutes les fonctions renvoient par défaut un `int` C. Pour préciser un autre type de sortie, il faut définir l'attribut `restype` de l'objet encapsulant la fonction.

Voici un exemple plus poussé. Celui-ci utilise la fonction `strchr`, qui prend en paramètres un pointeur vers une chaîne et un caractère. Elle renvoie un pointeur sur une chaîne de caractères :

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p      # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

Pour économiser l'appel `ord("x")`, il est possible de définir l'attribut `argtypes` ; le second argument, un objet octet à un seul caractère, sera automatiquement converti en un caractère C :

```

>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
>>>

```

Si la fonction à interfacier renvoie un entier, l'attribut `restype` peut aussi être un callable (une fonction ou une classe par exemple). Dans ce cas, l'appelable sera appelé avec l'entier renvoyé par la fonction et le résultat de cet appel sera le résultat final de l'appel à la fonction. C'est pratique pour vérifier les codes d'erreurs des valeurs de retour et lever automatiquement des exceptions :

```

>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>

```

`WinError` appelle l'API Windows `FormatMessage()` pour obtenir une représentation de la chaîne de caractères qui correspond au code d'erreur, et renvoie une exception. `WinError` prend en paramètre — optionnel — le code d'erreur. Si celui-ci n'est pas passé, elle appelle `GetLastError()` pour le récupérer.

Notez cependant que l'attribut `errcheck` permet de vérifier bien plus efficacement les erreurs ; référez-vous au manuel de référence pour plus de précisions.

Passage de pointeurs (passage de paramètres par référence)

Il arrive qu'une fonction C du code à interfacier requière un *pointeur* vers un certain type de donnée en paramètre, typiquement pour écrire à l'endroit correspondant ou si la donnée est trop grande pour pouvoir être passée par valeur. Ce mécanisme est appelé *passage de paramètres par référence*.

`ctypes` contient la fonction `byref()` qui permet de passer des paramètres par référence. La fonction `pointer()` a la même utilité, mais fait plus de travail car `pointer()` construit un véritable objet pointeur. Ainsi, si vous n'avez pas besoin de cet objet dans votre code Python, utiliser `byref()` est plus performant :

```

>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)

```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

Structures et unions

Les structures et les unions doivent hériter des classes de base *Structure* et *Union* définies dans le module *ctypes*. Chaque sous-classe doit définir un attribut `_fields_`. `_fields_` doit être une liste de *paires*, contenant un *nom de champ* et un *type de champ*.

Le type de champ doit être un type *ctypes* comme *c_int* ou un type *ctypes* dérivé : structure, union, tableau ou pointeur.

Voici un exemple simple : une structure POINT qui contient deux entiers *x* et *y* et qui montre également comment instancier une structure avec le constructeur :

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many initializers
>>>
```

Il est bien entendu possible de créer des structures plus complexes. Une structure peut elle-même contenir d'autres structures en prenant une structure comme type de champ.

Voici une structure RECT qui contient deux POINTs *upperleft* et *lowerright* :

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

Une structure encapsulée peut être instanciée par un constructeur de plusieurs façons :


```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Il est possible de récupérer les *descripteurs* des champs depuis la *classe*. Ils sont importants pour déboguer car ils contiennent des informations utiles :

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
>>>
```

Avertissement : *ctypes* ne prend pas en charge le passage par valeur des unions ou des structures avec des champs de bits. Bien que cela puisse fonctionner sur des architectures 32 bits avec un jeu d'instructions x86, ce n'est pas garanti par la bibliothèque en général. Les unions et les structures avec des champs de bits doivent toujours être passées par pointeur.

Alignement et boutisme des structures et des unions

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behavior by specifying a `_pack_` class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what `#pragma pack(n)` also does in MSVC.

ctypes suit le boutisme natif pour les *Structure* et les *Union*. Pour construire des structures avec un boutisme différent, utilisez les classes de base *BigEndianStructure*, *LittleEndianStructure*, *BigEndianUnion* ou *LittleEndianUnion*. Ces classes ne peuvent pas avoir de champ pointeur.

Champs de bits dans les structures et les unions

Il est possible de créer des structures et des unions contenant des champs de bits. Seuls les entiers peuvent être des champs de bits, le nombre de bits est défini dans le troisième champ du n-uplet `_fields_` :

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

Tableaux

Les tableaux sont des séquences qui contiennent un nombre fixe d'instances du même type.

La meilleure façon de créer des tableaux consiste à multiplier le type de donnée par un entier positif :

```
TenPointsArrayType = POINT * 10
```

Voici un exemple — un peu artificiel — d'une structure contenant, entre autres, 4 POINTs :

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

Comme d'habitude, on crée les instances en appelant la classe :

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

Le code précédent affiche une suite de 0 0 car le contenu du tableau est initialisé avec des zéros.

Des valeurs d'initialisation du bon type peuvent être passées :

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

Pointeurs

On crée une instance de pointeur en appelant la fonction `pointer()` sur un type `ctypes` :

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Les instances de pointeurs ont un attribut `contents` qui renvoie l'objet pointé (l'objet `i` ci-dessus) :

```
>>> pi.contents
c_long(42)
>>>
```

Attention, `ctypes` ne fait pas de ROI (retour de l'objet initial). Il crée un nouvel objet à chaque fois qu'on accède à un attribut :

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

Affecter une autre instance de `c_int` à l'attribut `contents` du pointeur fait pointer le pointeur vers l'adresse mémoire de cette nouvelle instance :

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

Il est possible d'indexer les pointeurs par des entiers :

```
>>> pi[0]
99
>>>
```

Affecter à travers un indice change la valeur pointée :

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

Si vous êtes sûr de vous, vous pouvez utiliser d'autres valeurs que 0, comme en C : il est ainsi possible de modifier une zone mémoire de votre choix. De manière générale cette fonctionnalité ne s'utilise que sur un pointeur renvoyé par une fonction C, pointeur que vous *savez* pointer vers un tableau et non sur un seul élément.

Sous le capot, la fonction `pointer()` fait plus que simplement créer une instance de pointeur ; elle doit d'abord créer un type « pointeur sur... ». Cela s'effectue avec la fonction `POINTER()`, qui prend en paramètre n'importe quel type `ctypes` et renvoie un nouveau type :

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

Appeler le pointeur sur type sans arguments crée un pointeur NULL. Les pointeurs NULL s'évaluent à `False` :

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

`ctypes` vérifie que le pointeur n'est pas NULL quand il en déréférence un (mais déréférencer des pointeurs non NULL invalides fait planter Python) :

```
>>> null_ptr[0]
Traceback (most recent call last):
....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
....
ValueError: NULL pointer access
>>>
```

Conversions de type

En général, `ctypes` respecte un typage fort. Cela signifie que si un `POINTER(c_int)` est présent dans la liste des `argtypes` d'une fonction ou est le type d'un attribut membre dans une définition de structure, seules des instances de ce type seront valides. Cette règle comporte quelques exceptions pour lesquelles `ctypes` accepte d'autres objets. Par exemple il est possible de passer des instances de tableau à place de pointeurs, s'ils sont compatibles. Dans le cas de `POINTER(c_int)`, `ctypes` accepte des tableaux de `c_int` :

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
2
3
>>>
```

De plus, si un paramètre de fonction est déclaré explicitement de type pointeur (comme `POINTER(c_int)`) dans les `argtypes`, il est aussi possible de passer un objet du type pointé — ici, `c_int` — à la fonction. `ctypes` appelle alors automatiquement la fonction de conversion `byref()`.

Pour mettre un champ de type `POINTER` à NULL, il faut lui affecter `None` :

```
>>> bar.values = None
>>>
```

Parfois il faut gérer des incompatibilités entre les types. En C, il est possible de convertir un type en un autre. `ctypes` fournit la fonction `cast()` qui permet la même chose. La structure `Bar` ci-dessus accepte des pointeurs `POINTER(c_int)` ou des tableaux de `c_int` comme valeur pour le champ `values`, mais pas des instances d'autres types :

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

C'est là que la fonction `cast()` intervient.

La fonction `cast()` permet de convertir une instance de *ctypes* en un pointeur vers un type de données *ctypes* différent. `cast()` prend deux paramètres : un objet *ctypes* qui est, ou qui peut être converti en, un certain pointeur et un type pointeur de *ctypes*. Elle renvoie une instance du second argument, qui pointe sur le même bloc mémoire que le premier argument :

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

Ainsi, la fonction `cast()` permet de remplir le champ `values` de la structure `Bar` :

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

Types incomplets

Un *type incomplet* est une structure, une union ou un tableau dont les membres ne sont pas encore définis. C'est l'équivalent d'une déclaration avancée en C, où la définition est fournie plus tard :

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

Une traduction naïve, mais invalide, en code *ctypes* ressemblerait à ça :

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

Cela ne fonctionne pas parce que la nouvelle `class cell` n'est pas accessible dans la définition de la classe elle-même. Dans le module *ctypes*, on définit la classe `cell` et on définira les `_fields_` plus tard, après avoir défini la classe :

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
>>>
```

Lets try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times :

```
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

Fonctions de rappel

`ctypes` permet de créer des pointeurs de fonctions appelables par des appelables Python. On les appelle parfois *fonctions de rappel*.

Tout d'abord, il faut créer une classe pour la fonction de rappel. La classe connaît la convention d'appel, le type de retour ainsi que le nombre et le type de paramètres que la fonction accepte.

La fabrique `CFUNCTYPE()` crée un type pour les fonctions de rappel qui suivent la convention d'appel `cdecl`. En Windows, c'est la fabrique `WINFUNCTYPE()` qui crée un type pour les fonctions de rappel qui suivent la convention d'appel `stdcall`.

Le premier paramètre de ces deux fonctions est le type de retour, et les suivants sont les types des arguments qu'attend la fonction de rappel.

Intéressons-nous à un exemple tiré de la bibliothèque standard C : la fonction `qsort()`. Celle-ci permet de classer des éléments par l'emploi d'une fonction de rappel. Nous allons utiliser `qsort()` pour ordonner un tableau d'entiers :

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` doit être appelée avec un pointeur vers la donnée à ordonner, le nombre d'éléments dans la donnée, la taille d'un élément et un pointeur vers le comparateur, c-à-d la fonction de rappel. Cette fonction sera invoquée avec deux pointeurs sur deux éléments et doit renvoyer un entier négatif si le premier élément est plus petit que le second, zéro s'ils sont égaux et un entier positif sinon.

Ainsi notre fonction de rappel reçoit des pointeurs vers des entiers et doit renvoyer un entier. Créons d'abord le type pour la fonction de rappel :

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

Pour commencer, voici une fonction de rappel simple qui affiche les valeurs qu'on lui passe :

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

Résultat :

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

À présent, comparons pour de vrai les deux entiers et renvoyons un résultat utile :

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

Et comme il est facile de le voir, notre tableau est désormais classé :

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

The function factories can be used as decorator factories, so we may as well write :

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

Note : Prenez garde à bien conserver une référence à un objet `CFUNCTYPE()` tant que celui-ci est utilisé par le code C. `ctypes` ne le fait pas tout seul et, si vous ne le faites pas, le ramasse-miette pourrait les libérer, ce qui fera planter votre programme quand un appel sera fait.

Notez aussi que si la fonction de rappel est appelée dans un fil d'exécution créé hors de Python (p. ex. par du code externe qui appelle la fonction de rappel), `ctypes` crée un nouveau fil Python « creux » à chaque fois. Ce comportement est acceptable pour la plupart des cas d'utilisation, mais cela implique que les valeurs stockées avec `threading.local` ne seront *pas* persistantes d'un appel à l'autre, même si les appels proviennent du même fil d'exécution C.

Accès aux variables exportées depuis une DLL

Certaines bibliothèques ne se contentent pas d'exporter des fonctions, elles exportent aussi des variables. Par exemple, la bibliothèque Python exporte `Py_OptimizeFlag`, un entier valant 0, 1, ou 2 selon que l'option `-O` ou `-OO` soit donnée au démarrage.

`ctypes` peut accéder à ce type de valeurs avec les méthodes de classe `in_dll()` du type considéré. `pythonapi` est un symbole prédéfini qui donne accès à l'API C Python :

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print(opt_flag)
c_long(0)
>>>
```

Si l'interpréteur est lancé avec `-O`, l'exemple affiche `c_long(1)` et `c_long(2)` avec `-OO`.

Le pointeur `PyImport_FrozenModules` exposé par Python est un autre exemple complet de l'utilisation de pointeurs.

Citons la documentation :

This pointer is initialized to point to an array of struct `_frozen` records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

Donc manipuler ce pointeur peut même se révéler utile. Pour limiter la taille de l'exemple, nous nous bornons à montrer comment lire ce tableau avec `ctypes` :

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

Le type de donnée struct `_frozen` ayant été défini, nous pouvons récupérer le pointeur vers le tableau :

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

Since `table` is a pointer to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the `NULL` entry :

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
__hello__ 161
__phello__ -161
__phello__.spam 161
>>>
```


The fact that standard Python has a frozen module and a frozen package (indicated by the negative size member) is not well known, it is only used for testing. Try it out with `import __hello__` for example.

Pièges

Il y a quelques cas tordus dans `ctypes` où on peut s'attendre à un résultat différent de la réalité.

Examinons l'exemple suivant :

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

Diantre. On s'attendait certainement à ce que le dernier résultat affiche 3 4 1 2. Que s'est-il passé ? Les étapes de la ligne `rc.a, rc.b = rc.b, rc.a` ci-dessus sont les suivantes :

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

Les objets `temp0` et `temp1` utilisent encore le tampon interne de l'objet `rc` ci-dessus. Donc exécuter `rc.a = temp0` copie le contenu du tampon de `temp0` dans celui de `rc`. Ce qui, par ricochet, modifie le contenu de `temp1`. Et donc, la dernière affectation, `rc.b = temp1`, n'a pas l'effet escompté.

Gardez en tête qu'accéder au sous-objet depuis une *Structure*, une *Union* ou un *Array* ne copie *pas* le sous-objet, mais crée un objet interface qui accède au tampon sous-jacent de l'objet initial.

Another example that may behave different from what one would expect is this :

```
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
>>>
```

Pourquoi cela affiche-t'il `False` ? Les instances `ctypes` sont des objets qui contiennent un bloc mémoire et des *descriptor* qui donnent accès au contenu du ce bloc. Stocker un objet Python dans le bloc mémoire ne stocke pas l'objet même ; seuls ses `contents` le sont. Accéder au `contents` crée un nouvel objet Python à chaque fois !

Types de données à taille flottante

`ctypes` assure la prise en charge des tableaux et des structures à taille flottante.

La fonction `resize()` permet de redimensionner la taille du tampon mémoire d'un objet `ctypes` existant. Cette fonction prend l'objet comme premier argument et la taille en octets désirée comme second. La taille du tampon mémoire ne peut pas être inférieure à celle occupée par un objet unitaire du type considéré. Une `ValueError` est levée si c'est le cas :

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

Cela dit, comment accéder aux éléments supplémentaires contenus dans le tableau ? Vu que le type ne connaît que 4 éléments, on obtient une erreur si l'on accède aux suivants :

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
...
IndexError: invalid index
>>>
```

Une autre approche pour utiliser des types de donnée à taille flottante avec `ctypes` consiste à tirer profit de la nature intrinsèquement dynamique de Python et de (re)définir le type de donnée une fois que la taille demandée est connue, au cas-par-cas.

16.16.2 Référence du module

Recherche de bibliothèques partagées

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the `find_library()` function is to locate a library in a way similar to what the compiler or runtime loader does (on platforms with several versions of a shared library the most recent should be loaded), while the `ctypes` library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

`ctypes.util.find_library(name)`

Try to find a library and return a pathname. *name* is the library name without any prefix like *lib*, suffix like *.so*, *.dylib* or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

On Linux, `find_library()` tries to run external programs (`/sbin/ldconfig`, `gcc`, `objdump` and `ld`) to find the library file. It returns the filename of the library file.

Modifié dans la version 3.6 : On Linux, the value of the environment variable `LD_LIBRARY_PATH` is used when searching for libraries, if a library cannot be found by any other means.

Here are some examples :

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

On OS X, `find_library()` tries several predefined naming schemes and paths to locate the library, and returns a full pathname if successful :

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

On Windows, `find_library()` searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with `ctypes`, it *may* be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using `find_library()` to locate the library at runtime.

Chargement des bibliothèques partagées

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes :

class `ctypes.CDLL` (*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*,
use_last_error=False)
 Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

class `ctypes.OleDLL` (*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*,
use_last_error=False)
 Windows only : Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific `HRESULT` code. `HRESULT` values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an `OSError` is automatically raised.
 Modifié dans la version 3.3 : `WindowsError` était levée auparavant.

class `ctypes.WinDLL` (*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*,
use_last_error=False)
 Windows only : Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.

On Windows CE only the standard calling convention is used, for convenience the *WinDLL* and *OleDLL* use the standard calling convention on this platform.

The Python *global interpreter lock* is released before calling any function exported by these libraries, and reacquired afterwards.

class `ctypes.PyDLL` (*name*, *mode*=*DEFAULT_MODE*, *handle*=*None*)

Instances of this class behave like *CDLL* instances, except that the Python GIL is *not* released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

Thus, this is only useful to call Python C api functions directly.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the *handle* named parameter, otherwise the underlying platforms *dlopen* or *LoadLibrary* function is used to load the library into the process, and to get a handle to it.

The *mode* parameter can be used to specify how the library is loaded. For details, consult the *dlopen*(3) manpage. On Windows, *mode* is ignored. On posix systems, *RTLD_NOW* is always added, and is not configurable.

The *use_errno* parameter, when set to true, enables a ctypes mechanism that allows accessing the system *errno* error number in a safe way. *ctypes* maintains a thread-local copy of the systems *errno* variable; if you call foreign functions created with *use_errno*=True then the *errno* value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

The function *ctypes.get_errno()* returns the value of the ctypes private copy, and the function *ctypes.set_errno()* changes the ctypes private copy to a new value and returns the former value.

The *use_last_error* parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the *GetLastError()* and *SetLastError()* Windows API functions; *ctypes.get_last_error()* and *ctypes.set_last_error()* are used to request and change the ctypes private copy of the windows error code.

`ctypes.RTLD_GLOBAL`

Flag to use as *mode* parameter. On platforms where this flag is not available, it is defined as the integer zero.

`ctypes.RTLD_LOCAL`

Flag to use as *mode* parameter. On platforms where this is not available, it is the same as *RTLD_GLOBAL*.

`ctypes.DEFAULT_MODE`

The default mode which is used to load shared libraries. On OSX 10.3, this is *RTLD_GLOBAL*, otherwise it is the same as *RTLD_LOCAL*.

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time :

```
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

The following public attributes are available, their name starts with an underscore to not clash with exported function names :

`PyDLL._handle`

The system handle used to access the library.

`PyDLL._name`

The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the `LibraryLoader` class, either by calling the `LoadLibrary()` method, or by retrieving the library as attribute of the loader instance.

class `ctypes.LibraryLoader` (*dlltype*)

Class which loads shared libraries. *dlltype* should be one of the `CDLL`, `PyDLL`, `WinDLL`, or `OleDLL` types.

`__getattr__()` has special behavior : It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

LoadLibrary (*name*)

Load a shared library into the process and return it. This method always returns a new instance of the library.

These prefabricated library loaders are available :

`ctypes.cdll`

Creates `CDLL` instances.

`ctypes.windll`

Windows only : Creates `WinDLL` instances.

`ctypes.oledll`

Windows only : Creates `OleDLL` instances.

`ctypes.pydll`

Creates `PyDLL` instances.

For accessing the C Python api directly, a ready-to-use Python shared library object is available :

`ctypes.pythonapi`

An instance of `PyDLL` that exposes Python C API functions as attributes. Note that all these functions are assumed to return C `int`, which is of course not always the truth, so you have to assign the correct `restype` attribute to use these functions.

Fonctions externes

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any `ctypes` data instances as arguments, and return the default result type specified by the library loader. They are instances of a private class :

class `ctypes._FuncPtr`

Base class for C callable foreign functions.

Instances of foreign functions are also C compatible data types ; they represent C function pointers.

This behavior can be customized by assigning to special attributes of the foreign function object.

restype

Assign a `ctypes` type to specify the result type of the foreign function. Use `None` for `void`, a function not returning anything.

It is possible to assign a callable Python object that is not a `ctypes` type, in this case the function is assumed to return a C `int`, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a `ctypes` data type as *restype* and assign a callable to the *errcheck* attribute.

argtypes

Assign a tuple of `ctypes` types to specify the argument types that the function accepts. Functions using the `stdcall` calling convention can only be called with the same number of arguments as the length of this tuple ; functions using the C calling convention accept additional, unspecified arguments as well.

When a foreign function is called, each actual argument is passed to the `from_param()` class method of the items in the *argtypes* tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a `c_char_p` item in the *argtypes* tuple will convert a string passed as argument into a bytes object using `ctypes` conversion rules.

New : It is now possible to put items in `argtypes` which are not `ctypes` types, but each item must have a `from_param()` method which returns a value usable as argument (integer, string, `ctypes` instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck

Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments :

callable (*result, func, arguments*)

result is what the foreign function returns, as specified by the `restype` attribute.

func is the foreign function object itself, this allows reusing the same callable object to check or post process the results of several functions.

arguments is a tuple containing the parameters originally passed to the function call, this allows specializing the behavior on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

exception `ctypes.ArgumentError`

This exception is raised when a foreign function call cannot convert one of the passed arguments.

Prototypes de fonction

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function, and can be used as decorator factories, and as such, be applied to functions through the `@wrapper` syntax. See *Fonctions de rappel* for examples.

`ctypes.CFUNCTYPE` (*restype, *argtypes, use_errno=False, use_last_error=False*)

The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If *use_errno* is set to true, the `ctypes` private copy of the system `errno` variable is exchanged with the real `errno` value before and after the call; *use_last_error* does the same for the Windows error code.

`ctypes.WINFUNCTYPE` (*restype, *argtypes, use_errno=False, use_last_error=False*)

Windows only : The returned function prototype creates functions that use the `stdcall` calling convention, except on Windows CE where `WINFUNCTYPE()` is the same as `CFUNCTYPE()`. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

`ctypes.PYFUNCTYPE` (*restype, *argtypes*)

The returned function prototype creates functions that use the Python calling convention. The function will *not* release the GIL during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call :

prototype (*address*)

Returns a foreign function at the specified address which must be an integer.

prototype (*callable*)

Create a C callable function (a callback function) from a Python *callable*.

prototype (*func_spec[, paramflags]*)

Returns a foreign function exported by a shared library. *func_spec* must be a 2-tuple (*name_or_ordinal, library*). The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

prototype (*vtbl_index, name[, paramflags[, iid]]*)

Returns a foreign function that will call a COM method. *vtbl_index* is the index into the virtual function table, a small non-negative integer. *name* is name of the COM method. *iid* is an optional pointer to the interface identifier which is used in extended error reporting.

COM methods use a special calling convention : They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the `argtypes` tuple.

The optional *paramflags* parameter creates foreign function wrappers with much more functionality than the features described above.

paramflags must be a tuple of the same length as *argtypes*.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter :

- 1 Specifies an input parameter to the function.
- 2 Output parameter. The foreign function fills in a value.
- 4 Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

This example demonstrates how to wrap the Windows `MessageBoxW` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this :

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

Here is the wrapping with *ctypes* :

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from ctypes"), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

The `MessageBox` foreign function can now be called in these ways :

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

A second example demonstrates output parameters. The win32 `GetWindowRect` function retrieves the dimensions of a specified window by copying them into `RECT` structure that the caller has to supply. Here is the C declaration :

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
    HWND hWnd,
    LPRECT lpRect);
```

Here is the wrapping with *ctypes* :

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the `GetWindowRect` function now returns a `RECT` instance, when called.

Output parameters can be combined with the `errcheck` protocol to do further output processing and error checking. The win32 `GetWindowRect` api function returns a `BOOL` to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed :

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

If the `errcheck` function returns the argument tuple it receives unchanged, `ctypes` continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a `RECT` instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place :

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

Fonctions utilitaires

`ctypes.addressof(obj)`

Returns the address of the memory buffer as integer. *obj* must be an instance of a ctypes type.

`ctypes.alignment(obj_or_type)`

Returns the alignment requirements of a ctypes type. *obj_or_type* must be a ctypes type or instance.

`ctypes.byref(obj[, offset])`

Returns a light-weight pointer to *obj*, which must be an instance of a ctypes type. *offset* defaults to zero, and must be an integer that will be added to the internal pointer value.

`byref(obj, offset)` corresponds to this C code :

```
((char *)&obj) + offset)
```

The returned object can only be used as a foreign function call parameter. It behaves similar to `pointer(obj)`, but the construction is a lot faster.

`ctypes.cast(obj, type)`

This function is similar to the cast operator in C. It returns a new instance of *type* which points to the same memory block as *obj*. *type* must be a pointer type, and *obj* must be an object that can be interpreted as a pointer.

`ctypes.create_string_buffer(init_or_size, size=None)`

This function creates a mutable character buffer. The returned object is a ctypes array of `c_char`.

init_or_size must be an integer which specifies the size of the array, or a bytes object which will be used to initialize the array items.

If a bytes object is specified as first argument, the buffer is made one item larger than its length so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the bytes should not be used.

`ctypes.create_unicode_buffer` (*init_or_size*, *size=None*)

This function creates a mutable unicode character buffer. The returned object is a ctypes array of `c_wchar`.

init_or_size must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

`ctypes.DllCanUnloadNow` ()

Windows only : This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the `DllCanUnloadNow` function that the `_ctypes` extension dll exports.

`ctypes.DllGetClassObject` ()

Windows only : This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the `DllGetClassObject` function that the `_ctypes` extension dll exports.

`ctypes.util.find_library` (*name*)

Try to find a library and return a pathname. *name* is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

`ctypes.util.find_msvcrt` ()

Windows only : return the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, `None` is returned.

If you need to free memory, for example, allocated by an extension module with a call to the `free(void *)`, it is important that you use the function in the same library that allocated the memory.

`ctypes.FormatError` ([*code*])

Windows only : Returns a textual description of the error code *code*. If no error code is specified, the last error code is used by calling the Windows api function `GetLastError`.

`ctypes.GetLastError` ()

Windows only : Returns the last error code set by Windows in the calling thread. This function calls the Windows `GetLastError()` function directly, it does not return the ctypes-private copy of the error code.

`ctypes.get_errno` ()

Returns the current value of the ctypes-private copy of the system `errno` variable in the calling thread.

`ctypes.get_last_error` ()

Windows only : returns the current value of the ctypes-private copy of the system `LastError` variable in the calling thread.

`ctypes.memmove` (*dst*, *src*, *count*)

Same as the standard C `memmove` library function : copies *count* bytes from *src* to *dst*. *dst* and *src* must be integers or ctypes instances that can be converted to pointers.

`ctypes.memset` (*dst*, *c*, *count*)

Same as the standard C `memset` library function : fills the memory block at address *dst* with *count* bytes of value *c*. *dst* must be an integer specifying an address, or a ctypes instance.

`ctypes.POINTER` (*type*)

This factory function creates and returns a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. *type* must be a ctypes type.

`ctypes.pointer(obj)`

This function creates a new pointer instance, pointing to *obj*. The returned object is of the type `POINTER(type(obj))`.

Note : If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

`ctypes.resize(obj, size)`

This function resizes the internal memory buffer of *obj*, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the buffer.

`ctypes.set_errno(value)`

Set the current value of the ctypes-private copy of the system `errno` variable in the calling thread to *value* and return the previous value.

`ctypes.set_last_error(value)`

Windows only : set the current value of the ctypes-private copy of the system `LastError` variable in the calling thread to *value* and return the previous value.

`ctypes.sizeof(obj_or_type)`

Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof` operator.

`ctypes.string_at(address, size=-1)`

This function returns the C string starting at memory address *address* as a bytes object. If *size* is specified, it is used as *size*, otherwise the string is assumed to be zero-terminated.

`ctypes.WinError(code=None, descr=None)`

Windows only : this function is probably the worst-named thing in ctypes. It creates an instance of `OSError`. If *code* is not specified, `GetLastError` is called to determine the error code. If *descr* is not specified, `FormatError()` is called to get a textual description of the error.

Modifié dans la version 3.3 : An instance of `WindowsError` used to be created.

`ctypes.wstring_at(address, size=-1)`

This function returns the wide character string starting at memory address *address* as a string. If *size* is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

Types de données

class `ctypes._CData`

This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the `addressof()` helper function. Another instance variable is exposed as `_objects`; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the *metaclass*) :

from_buffer (*source*[, *offset*])

This method returns a ctypes instance that shares the buffer of the *source* object. The *source* object must support the writable buffer interface. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a `ValueError` is raised.

from_buffer_copy (*source*[, *offset*])

This method creates a ctypes instance, copying the buffer from the *source* object buffer which must be readable. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a `ValueError` is raised.

from_address (*address*)

This method returns a ctypes type instance using the memory specified by *address* which must be an integer.

from_param (*obj*)

This method adapts *obj* to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function's *argtypes* tuple; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns *obj* if that is an instance of the type. Some types accept other objects as well.

in_dll (*library*, *name*)

This method returns a ctypes type instance exported by a shared library. *name* is the name of the symbol that exports the data, *library* is the loaded shared library.

Common instance variables of ctypes data types :

_b_base_

Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The *_b_base_* read-only member is the root ctypes object that owns the memory block.

_b_needsfree_

This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

_objects

This member is either *None* or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.

Types de données fondamentaux

class ctypes._SimpleCData

This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. *_SimpleCData* is a subclass of *_CData*, so it inherits their methods and attributes. ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute :

value

This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character bytes object or string, for character pointer types it is a Python bytes object or string.

When the *value* attribute is retrieved from a ctypes instance, usually a new object is returned each time. *ctypes* does *not* implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a *restype* of *c_char_p*, you will always receive a Python bytes object, *not* a *c_char_p* instance.

Subclasses of fundamental data types do *not* inherit this behavior. So, if a foreign functions *restype* is a subclass of *c_void_p*, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the *value* attribute.

These are the fundamental ctypes data types :

class ctypes.c_byte

Represents the C `signed char` datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_char

Represents the C `char` datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class `ctypes.c_char_p`
Represents the C `char *` datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, `POINTER(c_char)` must be used. The constructor accepts an integer address, or a bytes object.

class `ctypes.c_double`
Represents the C double datatype. The constructor accepts an optional float initializer.

class `ctypes.c_longdouble`
Represents the C long double datatype. The constructor accepts an optional float initializer. On platforms where `sizeof(long double) == sizeof(double)` it is an alias to `c_double`.

class `ctypes.c_float`
Represents the C float datatype. The constructor accepts an optional float initializer.

class `ctypes.c_int`
Represents the C signed int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to `c_long`.

class `ctypes.c_int8`
Represents the C 8-bit signed int datatype. Usually an alias for `c_byte`.

class `ctypes.c_int16`
Represents the C 16-bit signed int datatype. Usually an alias for `c_short`.

class `ctypes.c_int32`
Represents the C 32-bit signed int datatype. Usually an alias for `c_int`.

class `ctypes.c_int64`
Represents the C 64-bit signed int datatype. Usually an alias for `c_longlong`.

class `ctypes.c_long`
Represents the C signed long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class `ctypes.c_longlong`
Represents the C signed long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class `ctypes.c_short`
Represents the C signed short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class `ctypes.c_size_t`
Represents the C `size_t` datatype.

class `ctypes.c_ssize_t`
Represents the C `ssize_t` datatype.
Nouveau dans la version 3.2.

class `ctypes.c_ubyte`
Represents the C unsigned char datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class `ctypes.c_uint`
Represents the C unsigned int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for `c_ulong`.

class `ctypes.c_uint8`
Represents the C 8-bit unsigned int datatype. Usually an alias for `c_ubyte`.

class `ctypes.c_uint16`
Represents the C 16-bit unsigned `int` datatype. Usually an alias for `c_ushort`.

class `ctypes.c_uint32`
Represents the C 32-bit unsigned `int` datatype. Usually an alias for `c_uint`.

class `ctypes.c_uint64`
Represents the C 64-bit unsigned `int` datatype. Usually an alias for `c_ulonglong`.

class `ctypes.c_ulong`
Represents the C unsigned `long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class `ctypes.c_ulonglong`
Represents the C unsigned `long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class `ctypes.c_ushort`
Represents the C unsigned `short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class `ctypes.c_void_p`
Represents the C `void *` type. The value is represented as integer. The constructor accepts an optional integer initializer.

class `ctypes.c_wchar`
Represents the C `wchar_t` datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class `ctypes.c_wchar_p`
Represents the C `wchar_t *` datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class `ctypes.c_bool`
Represent the C `bool` datatype (more accurately, `_Bool` from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

class `ctypes.HRESULT`
Windows only : Represents a `HRESULT` value, which contains success or error information for a function or method call.

class `ctypes.py_object`
Represents the C `PyObject *` datatype. Calling this without an argument creates a `NULL PyObject *` pointer.

The `ctypes.wintypes` module provides quite some other Windows specific data types, for example `HWND`, `WPARAM`, or `DWORD`. Some useful structures like `MSG` or `RECT` are also defined.

Types de donnée dérivés de Structure

class `ctypes.Union(*args, **kw)`
Abstract base class for unions in native byte order.

class `ctypes.BigEndianStructure(*args, **kw)`
Abstract base class for structures in *big endian* byte order.

class `ctypes.LittleEndianStructure(*args, **kw)`
Abstract base class for structures in *little endian* byte order.

Structures with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.Structure(*args, **kw)

Abstract base class for structures in *native* byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a `__fields__` class variable. *ctypes* will create *descriptors* which allow reading and writing the fields by direct attribute accesses. These are the

`__fields__`

A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any ctypes data type.

For integer type fields like `c_int`, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the `__fields__` class variable *after* the class statement that defines the Structure subclass, this allows creating data types that directly or indirectly reference themselves :

```
class List(Structure):
    pass
List.__fields__ = [("pNext", POINTER(List)),
                  ...
                  ]
```

The `__fields__` class variable must, however, be defined before the type is first used (an instance is created, `sizeof()` is called on it, and so on). Later assignments to the `__fields__` class variable will raise an `AttributeError`.

It is possible to defined sub-subclasses of structure types, they inherit the fields of the base class plus the `__fields__` defined in the sub-subclass, if any.

`__pack__`

An optional small integer that allows overriding the alignment of structure fields in the instance. `__pack__` must already be defined when `__fields__` is assigned, otherwise it will have no effect.

`__anonymous__`

An optional sequence that lists the names of unnamed (anonymous) fields. `__anonymous__` must be already defined when `__fields__` is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. *ctypes* will create descriptors in the structure type that allows accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows) :

```
class _U(Union):
    __fields__ = [("lptdesc", POINTER(TYPEDESC)),
                  ("lpadesc", POINTER(ARRAYDESC)),
                  ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    __anonymous__ = ("u",)
    __fields__ = [("u", _U),
                  ("vt", VARTYPE)]
```

The TYPEDESC structure describes a COM data type, the `vt` field specifies which one of the union fields is valid. Since the `u` field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. `td.lptdesc` and `td.u.lptdesc` are equivalent, but the former is faster since it does not need to create a temporary union instance :

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to defined sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate `__fields__` variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they appear in `__fields__`. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize `__fields__` with the same name, or create new attributes for names not present in `__fields__`.

Tableaux et pointeurs

class `ctypes.Array(*args)`

Classe de base abstraite pour les *arrays*.

The recommended way to create concrete array types is by multiplying any *ctypes* data type with a positive integer. Alternatively, you can subclass this type and define `__length__` and `__type__` class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is *not* itself an *Array*.

`__length__`

A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an *IndexError*. Will be returned by `len()`.

`__type__`

Spécifie le type de chaque élément de l'*array*.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

class `ctypes._Pointer`

Private, abstract base class for pointers.

Concrete pointer types are created by calling `POINTER()` with the type that will be pointed to; this is done automatically by `pointer()`.

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so `len()` will raise *TypeError*. Negative subscripts will read from the memory *before* the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you're lucky).

`__type__`

Specifies the type pointed to.

contents

Returns the object to which to pointer points. Assigning to this attribute changes the pointer to point to the assigned object.

Exécution concurrente

Les modules documentés dans ce chapitre fournissent des outils d'exécution concurrente de code. Le choix de l'outil approprié dépend de la tâche à exécuter (limitée par le CPU (*CPU bound*), ou limitée la vitesse des entrées-sorties (*IO bound*)) et du style de développement désiré (coopération gérée par des événements ou multitâche préemptif). En voici un survol :

17.1 `threading` — Parallélisme basé sur les fils d'exécution (*threads*)

Code source : [Lib/threading.py](#)

Ce module élabore des interfaces haut-niveau de fils d'exécutions multiples (*threading*) conçues en s'appuyant sur le module bas-niveau `_thread`. Voir aussi le module `queue`.

The `dummy_threading` module is provided for situations where `threading` cannot be used because `_thread` is missing.

Note : Bien qu'ils ne soient pas listés ci-dessous, ce module gère toujours les noms en `camelCase` utilisés pour certaines méthodes et fonctions de ce module dans la série Python 2.x.

Ce module définit les fonctions suivantes :

`threading.active_count()`

Renvoie le nombre d'objets `Thread` actuellement vivants. Le compte renvoyé est égal à la longueur de la liste renvoyée par `enumerate()`.

`threading.current_thread()`

Renvoie l'objet `Thread` courant, correspondant au fil de contrôle de l'appelant. Si le fil de contrôle de l'appelant n'a pas été créé via le module `Thread`, un objet `thread` factice aux fonctionnalités limitées est renvoyé.

`threading.get_ident()`

Renvoie l'« identifiant de fil » du fil d'exécution courant. C'est un entier non nul. Sa valeur n'a pas de signification directe ; il est destiné à être utilisé comme valeur magique opaque, par exemple comme clef de dictionnaire de données pour chaque fil. Les identificateurs de fils peuvent être recyclés lorsqu'un fil se termine et qu'un autre fil est créé.

Nouveau dans la version 3.3.

`threading.enumerate()`

Renvoie une liste de tous les objets fil d'exécution *Thread* actuellement vivants. La liste inclut les fils démons, les fils factices créés par `current_thread()` et le fil principal. Elle exclut les fils terminés et les fils qui n'ont pas encore été lancés.

`threading.main_thread()`

Renvoie l'objet fil d'exécution *Thread* principal. Dans des conditions normales, le fil principal est le fil à partir duquel l'interpréteur Python a été lancé.

Nouveau dans la version 3.4.

`threading.settrace(func)`

Attache une fonction de traçage pour tous les fils d'exécution démarrés depuis le module *Thread*. La fonction *func* est passée à `sys.settrace()` pour chaque fil, avant que sa méthode `run()` soit appelée.

`threading.setprofile(func)`

Attache une fonction de profilage pour tous les fils d'exécution démarrés depuis le module *Threading*. La fonction *func* est passée à `sys.setprofile()` pour chaque fil, avant que sa méthode `run()` soit appelée.

`threading.stack_size([size])`

Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a *RuntimeError* is raised. If the specified stack size is invalid, a *ValueError* is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common ; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability : Windows, systems with POSIX threads.

Ce module définit également la constante suivante :

`threading.TIMEOUT_MAX`

La valeur maximale autorisée pour le paramètre *timeout* des fonctions bloquantes (`Lock.acquire()`, `RLock.acquire()`, `Condition.wait()`, etc.). Spécifier un délai d'attente supérieur à cette valeur lève une *OverflowError*.

Nouveau dans la version 3.2.

Ce module définit un certain nombre de classes, qui sont détaillées dans les sections ci-dessous.

La conception de ce module est librement basée sur le modèle des fils d'exécution de Java. Cependant, là où Java fait des verrous et des variables de condition le comportement de base de chaque objet, ils sont des objets séparés en Python. La classe Python *Thread* prend en charge un sous-ensemble du comportement de la classe *Thread* de Java ; actuellement, il n'y a aucune priorité, aucun groupe de fils d'exécution, et les fils ne peuvent être détruits, arrêtés, suspendus, repris ni interrompus. Les méthodes statiques de la classe *Thread* de Java, lorsqu'elles sont implémentées, correspondent à des fonctions au niveau du module.

Toutes les méthodes décrites ci-dessous sont exécutées de manière atomique.

17.1.1 Données locales au fil d'exécution

Les données locales au fil d'exécution (*thread-local data*) sont des données dont les valeurs sont propres à chaque fil. Pour gérer les données locales au fil, il suffit de créer une instance de `local` (ou une sous-classe) et d'y stocker des données :

```
mydata = threading.local()
mydata.x = 1
```

Les valeurs dans l'instance sont différentes pour des *threads* différents.

class `threading.local`

Classe qui représente les données locales au fil d'exécution.

Pour plus de détails et de nombreux exemples, voir la chaîne de documentation du module `_threading_local`.

17.1.2 Objets *Threads*

La classe fil d'exécution `Thread` représente une activité qui est exécutée dans un fil d'exécution séparé. Il y a deux façons de spécifier l'activité : en passant un objet callable au constructeur, ou en ré-implémentant la méthode `run()` dans une sous-classe. Aucune autre méthode (à l'exception du constructeur) ne doit être remplacée dans une sous-classe. En d'autres termes, réimplémentez *seulement* les méthodes `__init__()` et `run()` de cette classe.

Une fois qu'un objet fil d'exécution est créé, son activité doit être lancée en appelant la méthode `start()` du fil. Ceci invoque la méthode `run()` dans un fil d'exécution séparé.

Une fois que l'activité du fil d'exécution est lancée, le fil est considéré comme « vivant ». Il cesse d'être vivant lorsque sa méthode `run()` se termine – soit normalement, soit en levant une exception non gérée. La méthode `is_alive()` teste si le fil est vivant.

D'autres fils d'exécution peuvent appeler la méthode `join()` d'un fil. Ceci bloque le fil appelant jusqu'à ce que le fil dont la méthode `join()` est appelée soit terminé.

Un fil d'exécution a un nom. Le nom peut être passé au constructeur, et lu ou modifié via l'attribut `name`.

Un fil d'exécution peut être marqué comme « fil démon ». Un programme Python se termine quand il ne reste plus que des fils démons. La valeur initiale est héritée du fil d'exécution qui l'a créé. Cette option peut être définie par la propriété `daemon` ou par l'argument `daemon` du constructeur.

Note : Les fils d'exécution démons sont brusquement terminés à l'arrêt du programme Python. Leurs ressources (fichiers ouverts, transactions de base de données, etc.) peuvent ne pas être libérées correctement. Si vous voulez que vos fils s'arrêtent proprement, faites en sorte qu'ils ne soient pas démoniques et utilisez un mécanisme de signalisation approprié tel qu'un objet événement `Event`.

Il y a un objet « fil principal », qui correspond au fil de contrôle initial dans le programme Python. Ce n'est pas un fil démon.

Il y a une possibilité que des objets fil d'exécution « fictifs » soient créés. Ce sont des objets correspondant à des fils d'exécution « étrangers », qui sont des fils de contrôle démarrés en dehors du module de `threading`, par exemple directement depuis du code C. Les objets fils d'exécution fictifs ont des fonctionnalités limitées ; ils sont toujours considérés comme vivants et démoniques, et ne peuvent pas être attendus via `join()`. Ils ne sont jamais supprimés, car il est impossible de détecter la fin des fils d'exécution étrangers.

class `threading.Thread`(*group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None*)

Ce constructeur doit toujours être appelé avec des arguments nommés. Les arguments sont :

group doit être `None` ; cet argument est réservé pour une extension future lorsqu'une classe `ThreadGroup` sera implémentée.

target est l'objet callable qui doit être invoqué par la méthode `run()`. La valeur par défaut est `None`, ce qui signifie que rien n'est appelé.

name est le nom du fil d'exécution. Par défaut, un nom unique est construit de la forme « Thread-*N* » où *N* est un petit nombre décimal.

args est le tuple d'arguments pour l'invocation de l'objet callable. La valeur par défaut est `()`.

kwargs est un dictionnaire d'arguments nommés pour l'invocation de l'objet callable. La valeur par défaut est `{}`. S'il ne vaut pas `None`, *daemon* définit explicitement si le fil d'exécution est démonique ou pas. S'il vaut `None` (par défaut), la valeur est héritée du fil courant.

Si la sous-classe réimplémente le constructeur, elle doit s'assurer d'appeler le constructeur de la classe de base (`Thread.__init__()`) avant de faire autre chose au fil d'exécution.

Modifié dans la version 3.3 : Ajout de l'argument *daemon*.

start()

Lance l'activité du fil d'exécution.

Elle ne doit être appelée qu'une fois par objet de fil. Elle fait en sorte que la méthode `run()` de l'objet soit invoquée dans un fil d'exécution.

Cette méthode lève une `RuntimeError` si elle est appelée plus d'une fois sur le même objet fil d'exécution.

run()

Méthode représentant l'activité du fil d'exécution.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the *target* argument, if any, with sequential and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

join(timeout=None)

Attend que le fil d'exécution se termine. Ceci bloque le fil appelant jusqu'à ce que le fil dont la méthode `join()` est appelée se termine – soit normalement, soit par une exception non gérée – ou jusqu'à ce que le délai optionnel *timeout* soit atteint.

Lorsque l'argument *timeout* est présent et ne vaut pas `None`, il doit être un nombre en virgule flottante spécifiant un délai pour l'opération en secondes (ou fractions de secondes). Comme `join()` renvoie toujours `None`, vous devez appeler `is_alive()` après `join()` pour déterminer si le délai a expiré – si le fil d'exécution est toujours vivant, c'est que l'appel à `join()` a expiré.

Lorsque l'argument *timeout* n'est pas présent ou vaut `None`, l'opération se bloque jusqu'à ce que le fil d'exécution se termine.

Un fil d'exécution peut être attendu via `join()` de nombreuses fois.

`join()` lève une `RuntimeError` si une tentative est faite pour attendre le fil d'exécution courant car cela conduirait à un interblocage (*deadlock* en anglais). Attendre via `join()` un fil d'exécution avant son lancement est aussi une erreur et, si vous tentez de le faire, lève la même exception.

name

Une chaîne de caractères utilisée à des fins d'identification seulement. Elle n'a pas de sémantique. Plusieurs fils d'exécution peuvent porter le même nom. Le nom initial est défini par le constructeur.

getName()

setName()

Anciens accesseur et mutateur pour *name*; utilisez plutôt ce dernier directement.

ident

« L'identificateur de fil d'exécution » de ce fil ou `None` si le fil n'a pas été lancé. C'est un entier non nul. Voyez également la fonction `get_ident()`. Les identificateurs de fils peuvent être recyclés lorsqu'un fil se termine et qu'un autre fil est créé. L'identifiant est disponible même après que le fil ait terminé.

is_alive()

Renvoie si le fil d'exécution est vivant ou pas.

Cette méthode renvoie `True` depuis juste avant le démarrage de la méthode `run()` et jusqu'à juste après la terminaison de la méthode `run()`. La fonction `enumerate()` du module renvoie une liste de tous les fils d'exécution vivants.

daemon

Booléen indiquant si ce fil d'exécution est un fil démon (`True`) ou non (`False`). Celui-ci doit être défini avant que `start()` ne soit appelé, sinon `RuntimeError` est levée. Sa valeur initiale est héritée du fil d'exécution créateur ; le fil principal n'est pas un fil démon et donc tous les fils créés dans ce fil principal ont par défaut la valeur `daemon = False`.

Le programme Python se termine lorsqu'il ne reste plus de fils d'exécution non-démons vivants.

isDaemon()**setDaemon()**

Anciens accesseur et mutateur pour `daemon` ; utilisez plutôt ce dernier directement.

CPython implementation detail : En CPython, en raison du verrou global de l'interpréteur (*Global Interpreter Lock*), un seul fil d'exécution peut exécuter du code Python à la fois (même si certaines bibliothèques orientées performance peuvent surmonter cette limitation). Si vous voulez que votre application fasse un meilleur usage des ressources de calcul des machines multi-cœurs, nous vous conseillons d'utiliser `multiprocessing` ou `concurrent.futures.ProcessPoolExecutor`. Néanmoins, les fils d'exécutions multiples restent un modèle approprié si vous souhaitez exécuter simultanément plusieurs tâches limitées par les performances des entrées-sorties.

17.1.3 Verrous

Un verrou primitif n'appartient pas à un fil d'exécution lorsqu'il est verrouillé. En Python, c'est actuellement la méthode de synchronisation la plus bas-niveau qui soit disponible, implémentée directement par le module d'extension `_thread`.

Un verrou primitif est soit « verrouillé » soit « déverrouillé ». Il est créé dans un état déverrouillé. Il a deux méthodes, `acquire()` et `release()`. Lorsque l'état est déverrouillé, `acquire()` verrouille et se termine immédiatement. Lorsque l'état est verrouillé, `acquire()` bloque jusqu'à ce qu'un appel à `release()` provenant d'un autre fil d'exécution le déverrouille. À ce moment `acquire()` le verrouille à nouveau et rend la main. La méthode `release()` ne doit être appelée que si le verrou est verrouillé, elle le déverrouille alors et se termine immédiatement. Déverrouiller un verrou qui n'est pas verrouillé provoque une `RuntimeError`.

Locks also support the *context management protocol*.

When more than one thread is blocked in `acquire()` waiting for the state to turn to unlocked, only one thread proceeds when a `release()` call resets the state to unlocked ; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

class threading.Lock

The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released ; any thread may release it.

Note that `Lock` is actually a factory function which returns an instance of the most efficient version of the concrete `Lock` class that is supported by the platform.

acquire(blocking=True, timeout=-1)

Acquiert un verrou, bloquant ou non bloquant.

When invoked with the *blocking* argument set to `True` (the default), block until the lock is unlocked, then set it to locked and return `True`.

When invoked with the *blocking* argument set to `False`, do not block. If a call with *blocking* set to `True` would block, return `False` immediately ; otherwise, set the lock to locked and return `True`.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. A *timeout* argument of `-1` specifies an unbounded wait. It is forbidden to specify a *timeout* when *blocking* is false.

The return value is `True` if the lock is acquired successfully, `False` if not (for example if the *timeout* expired).

Modifié dans la version 3.2 : Le paramètre *timeout* est nouveau.

Modifié dans la version 3.2 : Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

release()

Release a lock. This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a `RuntimeError` is raised.

Il n'y a pas de valeur de retour.

17.1.4 RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of « owning thread » and « recursion level » in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock ; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its `acquire()` method ; this returns once the thread owns the lock. To unlock the lock, a thread calls its `release()` method. `acquire()/release()` call pairs may be nested ; only the final `release()` (the `release()` of the outermost pair) resets the lock to unlocked and allows another thread blocked in `acquire()` to proceed.

Reentrant locks also support the *context management protocol*.

class `threading.RLock`

This class implements reentrant lock objects. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking ; the thread must release it once for each time it has acquired it.

Note that `RLock` is actually a factory function which returns an instance of the most efficient version of the concrete `RLock` class that is supported by the platform.

acquire (*blocking=True, timeout=-1*)

Acquiert un verrou, bloquant ou non bloquant.

When invoked without arguments : if this thread already owns the lock, increment the recursion level by one, and return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the *blocking* argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the *blocking* argument set to false, do not block. If a call without an argument would block, return false immediately ; otherwise, do the same thing as when called without arguments, and return true.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. Return true if the lock has been acquired, false if the timeout has elapsed.

Modifié dans la version 3.2 : Le paramètre *timeout* est nouveau.

release()

Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A `RuntimeError` is raised if this method is called when the lock is unlocked.

Il n'y a pas de valeur de retour.

17.1.5 Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object : you don't have to track it separately.

A condition variable obeys the *context management protocol* : using the `with` statement acquires the associated lock for the duration of the enclosed block. The `acquire()` and `release()` methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The `wait()` method releases the lock, and then blocks until another thread awakens it by calling `notify()` or `notify_all()`. Once awakened, `wait()` re-acquires the lock and returns. It is also possible to specify a timeout.

The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notify_all()` method wakes up all threads waiting for the condition variable.

Note : the `notify()` and `notify_all()` methods don't release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notify_all()` finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notify_all()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity :

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

The `while` loop checking for the application's condition is necessary because `wait()` can return after an arbitrary long time, and the condition which prompted the `notify()` call may no longer hold true. This is inherent to multi-threaded programming. The `wait_for()` method can be used to automate the condition checking, and eases the computation of timeouts :

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

To choose between `notify()` and `notify_all()`, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class `threading.Condition` (*lock=None*)

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the *lock* argument is given and not `None`, it must be a `Lock` or `RLock` object, and it is used as the underlying lock. Otherwise, a new `RLock` object is created and used as the underlying lock.

Modifié dans la version 3.3 : changed from a factory function to a class.

acquire (*args)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock ; the return value is whatever that method returns.

release ()

Release the underlying lock. This method calls the corresponding method on the underlying lock ; there is no return value.

wait (timeout=None)

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notify_all()` call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the `timeout` argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an `RLock`, it is not released using its `release()` method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the `RLock` class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is `True` unless a given `timeout` expired, in which case it is `False`.

Modifié dans la version 3.2 : Previously, the method always returned `None`.

wait_for (predicate, timeout=None)

Wait until a condition evaluates to true. `predicate` should be a callable which result will be interpreted as a boolean value. A `timeout` may be provided giving the maximum time to wait.

This utility method may call `wait()` repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to `False` if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing :

```
while not predicate():
    cv.wait()
```

Therefore, the same rules apply as with `wait()` : The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held.

Nouveau dans la version 3.2.

notify (n=1)

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method wakes up at most `n` of the threads waiting for the condition variable ; it is a no-op if no threads are waiting.

The current implementation wakes up exactly `n` threads, if at least `n` threads are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than `n` threads.

Note : an awakened thread does not actually return from its `wait()` call until it can reacquire the lock. Since `notify()` does not release the lock, its caller should.

notify_all ()

Wake up all threads waiting on this condition. This method acts like `notify()`, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

17.1.6 Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names `P()` and `V()` instead of `acquire()` and `release()`).

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some other thread calls `release()`.

Semaphores also support the *context management protocol*.

class `threading.Semaphore` (*value=1*)

This class implements semaphore objects. A semaphore manages an atomic counter representing the number of `release()` calls minus the number of `acquire()` calls, plus an initial value. The `acquire()` method blocks if necessary until it can return without making the counter negative. If not given, *value* defaults to 1.

The optional argument gives the initial *value* for the internal counter; it defaults to 1. If the *value* given is less than 0, `ValueError` is raised.

Modifié dans la version 3.3 : changed from a factory function to a class.

acquire (*blocking=True*, *timeout=None*)

Acquire a semaphore.

When invoked without arguments :

- If the internal counter is larger than zero on entry, decrement it by one and return true immediately.
- If the internal counter is zero on entry, block until awoken by a call to `release()`. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return true. Exactly one thread will be awoken by each call to `release()`. The order in which threads are awoken should not be relied on.

When invoked with *blocking* set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with a *timeout* other than `None`, it will block for at most *timeout* seconds. If `acquire` does not complete successfully in that interval, return false. Return true otherwise.

Modifié dans la version 3.2 : Le paramètre *timeout* est nouveau.

release ()

Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

class `threading.BoundedSemaphore` (*value=1*)

Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, `ValueError` is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, *value* defaults to 1.

Modifié dans la version 3.3 : changed from a factory function to a class.

Semaphore Example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore :

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's `acquire` and `release` methods when they need to connect to the server :

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.

17.1.7 Event Objects

This is one of the simplest mechanisms for communication between threads : one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

class `threading.Event`

Class implementing event objects. An event manages a flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true. The flag is initially false.

Modifié dans la version 3.3 : changed from a factory function to a class.

is_set()

Return true if and only if the internal flag is true.

set()

Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call `wait()` once the flag is true will not block at all.

clear()

Reset the internal flag to false. Subsequently, threads calling `wait()` will block until `set()` is called to set the internal flag to true again.

wait(timeout=None)

Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls `set()` to set the flag to true, or until the optional timeout occurs.

When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

This method returns true if and only if the internal flag has been set to true, either before the wait call or after the wait starts, so it will always return `True` except if a timeout is given and the operation times out.

Modifié dans la version 3.1 : Previously, the method always returned `None`.

17.1.8 Timer Objects

This class represents an action that should be run only after a certain amount of time has passed — a timer. `Timer` is a subclass of `Thread` and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their `start()` method. The timer can be stopped (before its action has begun) by calling the `cancel()` method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

Par exemple :

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class `threading.Timer` (*interval, function, args=None, kwargs=None*)

Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed. If *args* is `None` (the default) then an empty list will be used. If *kwargs* is `None` (the default) then an empty dict will be used.

Modifié dans la version 3.3 : changed from a factory function to a class.

cancel ()

Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.

17.1.9 Barrier Objects

Nouveau dans la version 3.2.

This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the `wait()` method and will block until all of the threads have made their `wait()` calls. At this point, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread :

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class `threading.Barrier` (*parties, action=None, timeout=None*)

Create a barrier object for *parties* number of threads. An *action*, when provided, is a callable to be called by one of the threads when they are released. *timeout* is the default timeout value if none is specified for the `wait()` method.

wait (*timeout=None*)

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously. If a *timeout* is provided, it is used in preference to any that was supplied to the class constructor.

The return value is an integer in the range 0 to *parties* - 1, different for each thread. This can be used to select a thread to do some special housekeeping, e.g. :

```
i = barrier.wait()
if i == 0:
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Only one thread needs to print this
print("passed the barrier")
```

If an *action* was provided to the constructor, one of the threads will have called it prior to being released. Should this call raise an error, the barrier is put into the broken state.

If the call times out, the barrier is put into the broken state.

This method may raise a *BrokenBarrierError* exception if the barrier is broken or reset while a thread is waiting.

reset()

Return the barrier to the default, empty state. Any threads waiting on it will receive the *BrokenBarrierError* exception.

Note that using this function may require some external synchronization if there are other threads whose state is unknown. If a barrier is broken it may be better to just leave it and create a new one.

abort()

Put the barrier into a broken state. This causes any active or future calls to *wait()* to fail with the *BrokenBarrierError*. Use this for example if one of the needs to abort, to avoid deadlocking the application.

It may be preferable to simply create the barrier with a sensible *timeout* value to automatically guard against one of the threads going awry.

parties

The number of threads required to pass the barrier.

n_waiting

The number of threads currently waiting in the barrier.

broken

A boolean that is *True* if the barrier is in the broken state.

exception `threading.BrokenBarrierError`

This exception, a subclass of *RuntimeError*, is raised when the *Barrier* object is reset or broken.

17.1.10 Using locks, conditions, and semaphores in the `with` statement

All of the objects provided by this module that have *acquire()* and *release()* methods can be used as context managers for a `with` statement. The *acquire()* method will be called when the block is entered, and *release()* will be called when the block is exited. Hence, the following snippet :

```
with some_lock:
    # do something...
```

est équivalente à :

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

Currently, *Lock*, *RLock*, *Condition*, *Semaphore*, and *BoundedSemaphore* objects may be used as `with` statement context managers.

17.2 multiprocessing — Parallélisme par processus

Code source : [Lib/multiprocessing/](#)

17.2.1 Introduction

multiprocessing est un paquet qui permet l'instanciation de processus via la même API que le module *threading*. Le paquet *multiprocessing* offre à la fois des possibilités de programmation concurrente locale ou à distance, contournant les problèmes du *Global Interpreter Lock* en utilisant des processus plutôt que des fils d'exécution. Ainsi, le module *multiprocessing* permet au développeur de bénéficier entièrement des multiples processeurs sur une machine. Il tourne à la fois sur les systèmes Unix et Windows.

Le module *multiprocessing* introduit aussi des API sans analogues dans le module *threading*. Un exemple est l'objet *Pool* qui offre une manière pratique de paralléliser l'exécution d'une fonction sur de multiples valeurs d'entrée, distribuant ces valeurs entre les processus (parallélisme de données). L'exemple suivant présente la manière classique de définir une telle fonction dans un module afin que les processus fils puissent importer ce module avec succès. L'exemple basique de parallélisme de données utilise *Pool*,

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

affiche sur la sortie standard

```
[1, 4, 9]
```

La classe *Process*

Dans le module *multiprocessing*, les processus sont instanciés en créant un objet *Process* et en appelant sa méthode *start()*. La classe *Process* suit la même API que *threading.Thread*. Un exemple trivial d'un programme multi-processus est

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Pour afficher les IDs des processus impliqués, voici un exemple plus étoffé :

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

La nécessité de la ligne `if __name__ == '__main__':` est expliquée par *Lignes directrices de programmation*.

Contextes et méthodes de démarrage

Suivant la plateforme, *multiprocessing* gère trois manières de démarrer un processus. Ces *méthodes de démarrage* sont

spawn Le processus parent démarre un processus neuf avec un interpréteur Python. Le processus fils hérite uniquement des ressources nécessaires pour exécuter la méthode `run()` de l'objet associé au processus. En particulier, les descripteurs de fichiers superflus et gérés par le processus parent ne sont pas hérités. Démarrer un processus en utilisant cette méthode est plutôt lent par rapport à *fork* ou *forkserver*. Disponible sur Unix et Windows. Par défaut sur Windows.

fork Le processus parent utilise `os.fork()` pour *forker* l'interpréteur Python. Le processus fils, quand il démarre, est effectivement identique au processus parent. Toutes les ressources du parent sont héritées par le fils. Notez qu'il est problématique de *forker* sans danger un processus *multi-threadé*. Disponible uniquement sous Unix. Par défaut sous Unix.

forkserver Quand le programme démarre et choisit la méthode de démarrage *forkserver*, un processus serveur est lancé. Dès lors, chaque fois qu'un nouveau processus est nécessaire, le processus parent se connecte au serveur et lui demande de *forker* un nouveau processus. Le processus serveur de *fork* n'utilisant qu'un seul fil d'exécution, il peut utiliser `os.fork()` sans danger. Les ressources superflues ne sont pas héritées. Disponible sur les plateformes Unix qui acceptent le passage de descripteurs de fichiers à travers des tubes (*pipes*) Unix.

Modifié dans la version 3.4 : *spawn* ajouté à toutes les plateformes Unix, et *forkserver* ajouté à certaines plateformes Unix. Les processus fils n'héritent plus de tous les descripteurs héritables du parent sous Windows.

Sous Unix, utiliser les méthodes de démarrage *spawn* ou *forkserver* démarre aussi un processus *semaphore tracker* qui traque les sémaphores nommés non libérés créés par les processus du programme. Quand tous les processus sont terminés, le traqueur de sémaphores libère les sémaphores restants. Généralement il ne devrait pas y en avoir, mais si un processus a été tué par un signal, certains sémaphores ont pu « fuiter ». (Libérer les sémaphores nommés est une affaire sérieuse puisque le système n'en autorise qu'un certain nombre, et qu'ils ne seront pas automatiquement libérés avant le prochain redémarrage.)

Pour sélectionner une méthode de démarrage, utilisez la fonction `set_start_method()` dans la clause `if __name__ == '__main__':` du module principal. Par exemple :

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

`set_start_method()` ne doit pas être utilisée plus d’une fois dans le programme.

Alternativement, vous pouvez utiliser `get_context()` pour obtenir un contexte. Les contextes ont la même API que le module *multiprocessing*, et permettent l’utilisation de plusieurs méthodes de démarrage dans un même programme.

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Notez que les objets relatifs à un contexte ne sont pas forcément compatibles avec les processus d’un contexte différent. En particulier, les verrous créés avec le contexte *fork* ne peuvent pas être passés aux processus lancés avec les méthodes *spawn* ou *forkserver*.

Une bibliothèque qui veut utiliser une méthode de démarrage particulière devrait probablement faire appel à `get_context()` pour éviter d’interférer avec le choix de l’utilisateur de la bibliothèque.

Échange d’objets entre les processus

multiprocessing gère deux types de canaux de communication entre les processus :

Queues

La classe *Queue* est un clone assez proche de *queue.Queue*. Par exemple :

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

Les queues peuvent être utilisées par plusieurs fils d’exécution ou processus.

Tubes (*pipes*)

La fonction `Pipe()` renvoie une paire d'objets de connexion connectés à un tube qui est par défaut à double-sens. Par exemple :

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())    # prints "[42, None, 'hello']"
    p.join()
```

Les deux objets de connexion renvoyés par `Pipe()` représentent les deux bouts d'un tube. Chaque objet de connexion possède (entre autres) des méthodes `send()` et `recv()`. Notez que les données d'un tube peuvent être corrompues si deux processus (ou fils d'exécution) essaient de lire ou d'écrire sur le même bout du tube en même temps. Évidemment il n'y a pas de risque de corruption si les processus utilisent deux bouts différents en même temps.

Synchronisation entre processus

`multiprocessing` contient des équivalents à toutes les primitives de synchronisation de `threading`. Par exemple il est possible d'utiliser un verrou pour s'assurer qu'un seul processus à la fois écrit sur la sortie standard :

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Sans le verrou, les sorties des différents processus risquent d'être mélangées.

Partager un état entre les processus

Comme mentionné plus haut, il est généralement préférable d'éviter autant que possible d'utiliser des états partagés en programmation concurrente. C'est particulièrement vrai quand plusieurs processus sont utilisés.

Cependant, si vous devez réellement partager des données, `multiprocessing` permet de le faire de deux manières.

Mémoire partagée

Les données peuvent être stockées dans une mémoire partagée en utilisant des `Value` ou des `Array`. Par exemple, le code suivant


```

from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])

```

affiche

```

3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

```

Les arguments 'd' et 'i' utilisés à la création des `num` et `arr` sont des codes de types tels qu'utilisés par le module `array` : 'd' indique un flottant double-précision et 'i' indique un entier signé. Ces objets partagés seront sûr d'utilisation entre processus et fils d'exécution.

Pour plus de flexibilité dans l'utilisation de mémoire partagée, vous pouvez utiliser le module `multiprocessing.sharedctypes` qui permet la création d'objets arbitraires `ctypes` alloués depuis la mémoire partagée.

Processus serveur

Un objet gestionnaire renvoyé par `Manager()` contrôle un processus serveur qui détient les objets Python et autorise les autres processus à les manipuler à l'aide de mandataires.

Un gestionnaire renvoyé par `Manager()` supportera les types `list`, `dict`, `Namespace`, `Lock`, `RLock`, `Semaphore`, `BoundedSemaphore`, `Condition`, `Event`, `Barrier`, `Queue`, `Value` et `Array`. Par exemple,

```

from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)

```

affiche

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Les processus serveurs de questionnaires sont plus flexibles que les mémoires partagées parce qu'ils peuvent gérer des types d'objets arbitraires. Aussi, un questionnaire unique peut être partagé par les processus sur différentes machines à travers le réseau. Cependant, ils sont plus lents que les mémoires partagées.

Utiliser un réservoir de *workers*

La classe `Pool` représente une *pool* de processus de travail. Elle possède des méthodes qui permettent aux tâches d'être déchargées vers les processus de travail de différentes manières.

Par exemple :

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4, ..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))    # runs in *only* one process
        print(res.get(timeout=1))           # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 secs
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

Notez que les méthodes d'une *pool* ne devraient être utilisées que par le processus qui l'a créée.

Note : Fonctionnellement ce paquet exige que le module `__main__` soit importable par les fils. Cela est expliqué sur la page [Lignes directrices de programmation](#), il est cependant utile de le rappeler ici. Cela signifie que certains exemples, comme les exemples utilisant `multiprocessing.pool.Pool`, ne fonctionnent pas dans l'interpréteur interactif. Par exemple :

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(Si vous essayez ce code, il affichera trois traces d'appels complètes entrelacées de manière semi-aléatoire, et vous aurez alors à stopper le processus maître.)

17.2.2 Référence

Le paquet `multiprocessing` reproduit en grande partie l'API du module `threading`.

Process et exceptions

class `multiprocessing.Process` (*group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None*)

Les objets *process* représentent une activité exécutée dans un processus séparé. La classe *Process* a des équivalents à toutes les méthodes de *threading.Thread*.

Le constructeur doit toujours être appelé avec des arguments nommés. *group* devrait toujours être `None` ; il existe uniquement pour la compatibilité avec *threading.Thread*. *target* est l'objet callable qui est invoqué par la méthode `run()`. Il vaut `None` par défaut, signifiant que rien n'est appelé. *name* est le nom du processus (voir *name* pour plus de détails). *args* est le *tuple* d'arguments pour l'invocation de la cible. *kwargs* est le dictionnaire des arguments nommés pour l'invocation de la cible. S'il est fourni, l'argument nommé *daemon* met l'option *daemon* du processus à `True` ou `False`. S'il est `None` (par défaut), l'option est héritée par le processus créateur.

Par défaut, aucun argument n'est passé à *target*.

Si une sous-classe redéfinit le constructeur, elle doit s'assurer d'invoquer le constructeur de la classe de base (`Process.__init__()`) avant de faire autre chose du processus.

Modifié dans la version 3.3 : Ajout de l'argument *daemon*.

run()

Méthode représentant l'activité du processus.

Vous pouvez redéfinir cette méthode dans une sous-classe. La méthode standard `run()` invoque l'objet callable passé au constructeur comme argument *target*, si fourni, avec les arguments séquentiels et nommés respectivement pris depuis les paramètres *args* et *kwargs*.

start()

Démarre l'activité du processus.

Elle doit être appelée au plus une fois par objet processus. Elle s'arrange pour que la méthode `run()` de l'objet soit invoquée dans un processus séparé.

join([timeout])

Si l'argument optionnel `timeout` est `None` (par défaut), la méthode bloque jusqu'à ce que le processus dont la méthode `join()` a été appelée se termine. Si `timeout` est un nombre positif, elle bloque au maximum pendant `timeout` secondes. Notez que la méthode renvoie `None` si le processus se termine ou si le temps d'exécution expire. Vérifiez l'attribut `exitcode` du processus pour déterminer s'il s'est terminé.

`join` peut être appelée plusieurs fois sur un même processus.

Un processus ne peut pas s'attendre lui-même car cela causerait un interblocage. C'est une erreur d'essayer d'attendre un processus avant qu'il ne soit démarré.

name

Le nom du processus. Le nom est une chaîne de caractères utilisée uniquement pour l'identification du processus. Il n'a pas de sémantique. Plusieurs processus peuvent avoir le même nom.

Le nom initial est déterminé par le constructeur. Si aucun nom explicite n'est fourni au constructeur, un nom de la forme « Process- N_1 : N_2 :...: N_k » est construit, où chaque N_k est le N -ième enfant de son parent.

is_alive()

Renvoie vrai si le processus est en vie, faux sinon.

Grossièrement, un objet processus est en vie depuis le moment où la méthode `start()` finit de s'exécuter jusqu'à ce que le processus fils se termine.

daemon

L'option `daemon` du processus, une valeur booléenne. L'option doit être réglée avant que la méthode `start()` ne soit appelée.

La valeur initiale est héritée par le processus créateur.

Quand un processus se ferme, il tente de terminer tous ses processus enfants `daemon`.

Notez qu'un processus `daemon` n'est pas autorisé à créer des processus fils. Sinon un processus `daemon` laisserait ses enfants orphelins lorsqu'il se termine par la fermeture de son parent. De plus, ce **ne sont pas** des `daemons` ou services Unix, ce sont des processus normaux qui seront terminés (et non attendus) si un processus non `daemon` se ferme.

En plus de l'API `threading.Thread`, les objets `Process` supportent aussi les attributs et méthodes suivants :

pid

Renvoie l'ID du processus. Avant que le processus ne soit lancé, la valeur est `None`.

exitcode

Le code de fermeture de l'enfant. La valeur est `None` si le processus ne s'est pas encore terminé. Une valeur négative $-N$ indique que le fils a été terminé par un signal N .

authkey

La clé d'authentification du processus (une chaîne d'octets).

Quand `multiprocessing` est initialisé, une chaîne aléatoire est assignée au processus principal, en utilisant `os.urandom()`.

Quand un objet `Process` est créé, il hérite de la clé d'authentification de son parent, bien que cela puisse être changé à l'aide du paramètre `authkey` pour une autre chaîne d'octets.

Voir *Clés d'authentification*.

sentinel

Un identifiant numérique de l'objet système qui devient « prêt » quand le processus se termine.

Vous pouvez utiliser cette valeur si vous voulez attendre plusieurs événements à la fois en utilisant `multiprocessing.connection.wait()`. Autrement appeler `join()` est plus simple.

Sous Windows, c'est un mécanisme de l'OS utilisable avec les familles d'appels API `WaitForSingleObject` et `WaitForMultipleObjects`. Sous Unix, c'est un descripteur de fichier utilisable avec les primitives sur module `select`.

Nouveau dans la version 3.3.

terminate()

Termine le processus. Sous Unix cela est réalisé à l'aide d'un signal `SIGTERM`, sous Windows `TerminateProcess()` est utilisé. Notez que les gestionnaires de sortie, les clauses *finally* etc. ne sont pas exécutées.

Notez que les descendants du processus ne *seront pas* terminés – ils deviendront simplement orphelins.

Avertissement : Si cette méthode est utilisée quand le processus associé utilise un tube ou une queue, alors le tube ou la queue sont susceptibles d'être corrompus et peuvent devenir inutilisables par les autres processus. De façon similaire, si le processus a acquis un verrou, un sémaphore ou autre, alors le terminer est susceptible de provoquer des blocages dans les autres processus.

Notez que les méthodes `start()`, `join()`, `is_alive()`, `terminate()` et `exitcode` ne devraient être appelées que par le processus ayant créé l'objet `process`.

Exemple d'utilisation de quelques méthodes de `Process` :

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<Process(Process-1, initial)> False
>>> p.start()
>>> print(p, p.is_alive())
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process(Process-1, stopped[SIGTERM])> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.ProcessError

La classe de base de toutes les exceptions de `multiprocessing`.

exception multiprocessing.BufferTooShort

Exception levée par `Connection.recv_bytes_into()` quand l'objet tampon fourni est trop petit pour le message à lire.

Si `e` est une instance de `BufferTooShort` alors `e.args[0]` donnera un message sous forme d'une chaîne d'octets.

exception multiprocessing.AuthenticationError

Levée quand il y a une erreur d'authentification.

exception multiprocessing.TimeoutError

Levée par les méthodes avec temps d'exécution limité, quand ce temps expire.

Tubes (*pipes*) et Queues

Quand de multiples processus sont utilisés, de l'échange de messages est souvent mis en place pour la communication entre processus et éviter d'avoir à utiliser des primitives de synchronisation telles que des verrous.

Pour échanger des messages vous pouvez utiliser un `Pipe()` (pour une connexion entre deux processus) ou une queue (qui autorise de multiples producteurs et consommateurs).

Les types `Queue`, `SimpleQueue` et `JoinableQueue` sont des queues FIFO multi-producteurs et multi-consommateurs modelées sur la classe `queue.Queue` de la bibliothèque standard. Elles diffèrent par l'absence dans `Queue` des méthodes `task_done()` et `join()` introduites dans la classe `queue.Queue` par Python 2.5.

Si vous utilisez `JoinableQueue` alors vous **devez** appeler `JoinableQueue.task_done()` pour chaque tâche retirée de la queue, sans quoi le sémaphore utilisé pour compter le nombre de tâches non accomplies pourra éventuellement déborder, levant une exception.

Notez que vous pouvez aussi créer une queue partagée en utilisant un objet gestionnaire – voir *Gestionnaires*.

Note : `multiprocessing` utilise les exceptions habituelles `queue.Empty` et `queue.Full` pour signaler un dépassement du temps maximal autorisé. Elles ne sont pas disponibles dans l'espace de nommage `multiprocessing` donc vous devez les importer depuis le module `queue`.

Note : Quand un objet est placé dans une queue, l'objet est sérialisé par `pickle` et un fil d'exécution en arrière-plan transmettra ensuite les données sérialisées sur un tube sous-jacent. Cela a certaines conséquences qui peuvent être un peu surprenantes, mais ne devrait causer aucune difficultés pratiques – si elles vous embêtent vraiment, alors vous pouvez à la place utiliser une queue créée avec un *manager*.

- (1) Après avoir placé un objet dans une queue vide il peut y avoir un délai infinitésimal avant que la méthode `empty()` de la queue renvoie `False` et que `get_nowait()` renvoie une valeur sans lever de `queue.Empty`.
 - (2) Si plusieurs processus placent des objets dans la queue, il est possible pour les objets d'être reçus de l'autre côté dans le désordre. Cependant, les objets placés par un même processus seront toujours récupérés dans l'ordre attendu.
-

Avertissement : Si un processus est tué à l'aide de `Process.terminate()` ou `os.kill()` pendant qu'il tente d'utiliser une `Queue`, alors les données de la queue peuvent être corrompues. Cela peut par la suite causer des levées d'exceptions dans les autres processus quand ils tenteront d'utiliser la queue.

Avertissement : Comme mentionné plus haut, si un processus fils a placé des éléments dans la queue (et qu'il n'a pas utilisé `JoinableQueue.cancel_join_thread`), alors le processus ne se terminera pas tant que les éléments placés dans le tampon n'auront pas été transmis au tube.

Cela signifie que si vous essayez d'attendre ce processus vous pouvez obtenir un interblocage, à moins que vous ne soyez sûr que tous les éléments placés dans la queue ont été consommés. De même, si le processus fils n'est pas un *daemon* alors le processus parent pourrait bloquer à la fermeture quand il tentera d'attendre tous ses enfants non *daemons*.

Notez que la queue créée à l'aide d'un gestionnaire n'a pas ce problème. Voir *Lignes directrices de programmation*.

Pour un exemple d'utilisation de queues pour de la communication entre les processus, voir *Exemples*.

`multiprocessing.Pipe([duplex])`

Renvoie une paire (`conn1`, `conn2`) d'objets `Connection` représentant les bouts d'un tube.

Si `duplex` vaut `True` (par défaut), alors le tube est bidirectionnel. Si `duplex` vaut `False` il est unidirectionnel : `conn1` ne peut être utilisé que pour recevoir des messages et `conn2` que pour en envoyer.

`class multiprocessing.Queue([maxsize])`

Renvoie une queue partagée entre les processus utilisant un tube et quelques verrous/sémaphores. Quand un processus place initialement un élément sur la queue, un fil d'exécution *feeder* est démarré pour transférer les objets du tampon vers le tube.

Les exceptions habituelles `queue.Empty` et `queue.Full` du module `queue` de la bibliothèque standard sont levées pour signaler les *timeouts*.

`Queue` implémente toutes les méthodes de `queue.Queue` à l'exception de `task_done()` et `join()`.

qsize()

Renvoie la taille approximative de la queue. Ce nombre n'est pas fiable en raison des problématiques de *multithreading* et *multiprocessing*.

Notez que cela peut lever une *NotImplementedError* sous les plateformes Unix telles que Mac OS X où *sem_getvalue()* n'est pas implémentée.

empty()

Renvoie *True* si la queue est vide, *False* sinon. Cette valeur n'est pas fiable en raison des problématiques de *multithreading* et *multiprocessing*.

full()

Renvoie *True* si la queue est pleine, *False* sinon. Cette valeur n'est pas fiable en raison des problématiques de *multithreading* et *multiprocessing*.

put(obj[, block[, timeout]])

Place *obj* dans la queue. Si l'argument optionnel *block* vaut *True* (par défaut) est que *timeout* est *None* (par défaut), bloque jusqu'à ce qu'un slot libre soit disponible. Si *timeout* est un nombre positif, la méthode bloquera au maximum *timeout* secondes et lèvera une exception *queue.Full* si aucun slot libre n'a été trouvé dans le temps imparti. Autrement (*block* vaut *False*), place un élément dans la queue si un slot libre est immédiatement disponible, ou lève une exception *queue.Full* dans le cas contraire (*timeout* est ignoré dans ce cas).

put_nowait(obj)

Équivalent à *put(obj, False)*.

get([block[, timeout]])

Retire et renvoie un élément de la queue. Si l'argument optionnel *block* vaut *True* (par défaut) et que *timeout* est *None* (par défaut), bloque jusqu'à ce qu'un élément soit disponible. Si *timeout* (le délai maximal autorisé) est un nombre positif, la méthode bloquera au maximum *timeout* secondes et lèvera une exception *queue.Empty* si aucun élément n'est disponible dans le temps imparti. Autrement (*block* vaut *False*), renvoie un élément s'il est immédiatement disponible, ou lève une exception *queue.Empty* dans le cas contraire (*timeout* est ignoré dans ce cas).

get_nowait()

Équivalent à *get(False)*.

multiprocessing.Queue possède quelques méthodes additionnelles non présentes dans *queue.Queue*. Ces méthodes ne sont habituellement pas nécessaires pour la plupart des codes :

close()

Indique que plus aucune donnée ne peut être placée sur la queue par le processus courant. Le fil d'exécution en arrière-plan se terminera quand il aura transféré toutes les données du tampon vers le tube. Elle est appelée automatiquement quand la queue est collectée par le ramasse-miettes.

join_thread()

Attend le fil d'exécution d'arrière-plan. Elle peut seulement être utilisée une fois que *close()* a été appelée. Elle bloque jusqu'à ce que le fil d'arrière-plan se termine, assurant que toutes les données du tampon ont été transmises au tube.

Par défaut si un processus n'est pas le créateur de la queue alors à la fermeture elle essaiera d'attendre le fil d'exécution d'arrière-plan de la queue. Le processus peut appeler *cancel_join_thread()* pour que *join_thread()* ne fasse rien.

cancel_join_thread()

Empêche *join_thread()* de bloquer. En particulier, cela empêche le fil d'arrière-plan d'être attendu automatiquement quand le processus se ferme – voir *join_thread()*.

Un meilleur nom pour cette méthode pourrait être *allow_exit_without_flush()*. Cela peut provoquer des pertes de données placées dans la queue, et vous ne devriez certainement pas avoir besoin de l'utiliser. Elle n'est là que si vous souhaitez terminer immédiatement le processus sans transférer les données du tampon, et que vous ne vous inquiétez pas de perdre des données.

Note : Le fonctionnement de cette classe requiert une implémentation de sémaphore partagé sur le système d'exploitation hôte. Sans cela, la fonctionnalité sera désactivée et la tentative d'instancier une *Queue* lèvera une

ImportError. Voir [bpo-3770](#) pour plus d'informations. Cette remarque reste valable pour les autres types de queues spécialisées définies par la suite.

class multiprocessing.**SimpleQueue**

Un type de *Queue* simplifié, très proche d'un *Pipe* avec verrou.

empty()

Renvoie True si la queue est vide, False sinon.

get()

Supprime et renvoie un élément de la queue.

put(item)

Place *item* dans la queue.

class multiprocessing.**JoinableQueue**([*maxsize*])

JoinableQueue, une sous-classe de *Queue*, est une queue qui ajoute des méthodes *task_done()* et *join()*.

task_done()

Indique qu'une tâche précédemment placée dans la queue est complétée. Utilisé par les consommateurs de la queue. Pour chaque *get()* utilisé pour récupérer une tâche, un appel ultérieur à *task_done()* indique à la queue que le traitement de la tâche est terminé.

Si un *join()* est actuellement bloquant, il se débloquent quand tous les éléments auront été traités (signifiant qu'un appel à *task_done()* a été reçu pour chaque élément ayant été placé via *put()* dans la queue).

Lève une exception *ValueError* si appelée plus de fois qu'il y avait d'éléments dans la file.

join()

Bloque jusqu'à ce que tous les éléments de la queue aient été récupérés et traités.

Le compteur des tâches non accomplies augmente chaque fois qu'un élément est ajouté à la queue. Le compteur redescend chaque fois qu'un consommateur appelle *task_done()* pour indiquer qu'un élément a été récupéré et que tout le travail qui le concerne est complété. Quand le compteur des tâches non accomplies atteint zéro, *join()* est débloquée.

Divers

multiprocessing.**active_children()**

Renvoie la liste de tous les enfants vivants du processus courant.

Appeler cette méthode provoque l'effet de bord d'attendre tout processus qui n'a pas encore terminé.

multiprocessing.**cpu_count()**

Renvoie le nombre de CPU sur le système.

Ce nombre n'est pas équivalent au nombre de CPUs que le processus courant peut utiliser. Le nombre de CPUs utilisables peut être obtenu avec `len(os.sched_getaffinity(0))`

Peut lever une *NotImplementedError*.

Voir aussi :

os.cpu_count()

multiprocessing.**current_process()**

Renvoie l'objet *Process* correspondant au processus courant.

Un analogue à *threading.current_thread()*.

multiprocessing.**freeze_support()**

Ajoute le support des programmes utilisant *multiprocessing* qui ont été gelés pour produire un exécutable Windows. (Testé avec *py2exe*, *PyInstaller* et *cx_Freeze*.)

Cette fonction doit être appelée juste après la ligne `if __name__ == '__main__':` du module principal. Par exemple :


```

from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()

```

Si la ligne `freeze_support()` est omise, alors tenter de lancer l'exécutable gelé lèvera une *RuntimeError*. Appeler `freeze_support()` n'a pas d'effet quand elle est invoquée sur un système d'exploitation autre que Windows. De plus, si le module est lancé normalement par l'interpréteur Python sous Windows (le programme n'a pas été gelé), alors `freeze_support()` n'a pas d'effet.

`multiprocessing.get_all_start_methods()`

Renvoie la liste des méthodes de démarrage supportées, la première étant celle par défaut. Les méthodes de démarrage possibles sont 'fork', 'spawn' et 'forkserver'. Sous Windows seule 'spawn' est disponible. Sous Unix 'fork' et 'spawn' sont disponibles, 'fork' étant celle par défaut.

Nouveau dans la version 3.4.

`multiprocessing.get_context(method=None)`

Renvoie un contexte ayant les mêmes attributs que le module *multiprocessing*.

Si *method* est *None* le contexte par défaut est renvoyé. Sinon *method* doit valoir 'fork', 'spawn' ou 'forkserver'. Une *ValueError* est levée si la méthode de démarrage spécifiée n'est pas disponible.

Nouveau dans la version 3.4.

`multiprocessing.get_start_method(allow_none=False)`

Renvoie le nom de la méthode de démarrage utilisée pour démarrer le processus.

Si le nom de la méthode n'a pas été fixé et que *allow_none* est faux, alors la méthode de démarrage est réglée à celle par défaut et son nom est renvoyé. Si la méthode n'a pas été fixée et que *allow_none* est vrai, *None* est renvoyé.

La valeur de retour peut être 'fork', 'spawn', 'forkserver' ou *None*. 'fork' est la valeur par défaut sous Unix, 'spawn' est celle sous Windows.

Nouveau dans la version 3.4.

`multiprocessing.set_executable()`

Définit le chemin de l'interpréteur Python à utiliser pour démarrer un processus fils. (Par défaut *sys.executable* est utilisé). Les intégrateurs devront probablement faire quelque chose comme

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

avant de pouvoir créer des processus fils.

Modifié dans la version 3.4 : Maintenant supporté sous Unix quand la méthode de démarrage 'spawn' est utilisée.

`multiprocessing.set_start_method(method)`

Règle la méthode qui doit être utilisée pour démarrer un processus fils. *method* peut être 'fork', 'spawn' ou 'forkserver'.

Notez que cette fonction ne devrait être appelée qu'une fois au plus, et l'appel devrait être protégé à l'intérieur d'une clause `if __name__ == '__main__':` dans le module principal.

Nouveau dans la version 3.4.

Note : *multiprocessing* ne contient pas d'analogues à *threading.active_count()*, *threading.enumerate()*, *threading.settrace()*, *threading.setprofile()*, *threading.Timer*, ou *threading.local*.

Objets de connexions

Les objets de connexion autorisent l'envoi et la réception d'objets sérialisables ou de chaînes de caractères. Ils peuvent être vus comme des interfaces de connexion (*sockets*) connectées orientées messages.

Les objets de connexion sont habituellement créés via *Pipe* – voir aussi *Auditeurs et Clients*.

class multiprocessing.connection.Connection

send (*obj*)

Envoie un objet sur l'autre bout de la connexion, qui devra être lu avec *recv()*.

The object must be picklable. Very large pickles (approximately 32 MB+, though it depends on the OS) may raise a *ValueError* exception.

recv ()

Renvoie un objet envoyé depuis l'autre bout de la connexion en utilisant *send()*. Bloque jusqu'à ce que quelque chose soit reçu. Lève une *EOFError* s'il n'y a plus rien à recevoir et que l'autre bout a été fermé.

fileno ()

Renvoie le descripteur de fichier ou identifiant utilisé par la connexion.

close ()

Ferme la connexion.

Elle est appelée automatiquement quand la connexion est collectée par le ramasse-miettes.

poll ([*timeout*])

Renvoie vrai ou faux selon si des données sont disponibles à la lecture.

Si *timeout* n'est pas spécifié la méthode renverra immédiatement. Si *timeout* est un nombre alors il spécifie le temps maximum de blocage en secondes. Si *timeout* est *None*, un temps d'attente infini est utilisé.

Notez que plusieurs objets de connexions peuvent être attendus en même temps à l'aide de *multiprocessing.connection.wait()*.

send_bytes (*buffer*[, *offset*[, *size*]])

Envoie des données binaires depuis un *bytes-like object* comme un message complet.

If *offset* is given then data is read from that position in *buffer*. If *size* is given then that many bytes will be read from *buffer*. Very large buffers (approximately 32 MB+, though it depends on the OS) may raise a *ValueError* exception

recv_bytes ([*maxlength*])

Renvoie un message complet de données binaires envoyées depuis l'autre bout de la connexion comme une chaîne de caractères. Bloque jusqu'à ce qu'il y ait quelque chose à recevoir. Lève une *EOFError* s'il ne reste rien à recevoir et que l'autre côté de la connexion a été fermé.

Si *maxlength* est précisé que le message est plus long que *maxlength* alors une *OSError* est levée et la connexion n'est plus lisible.

Modifié dans la version 3.3 : Cette fonction levait auparavant une *IOError*, qui est maintenant un alias pour *OSError*.

recv_bytes_into (*buffer*[, *offset*])

Lit et stocke dans *buffer* un message complet de données binaires envoyées depuis l'autre bout de la connexion et renvoie le nombre d'octets du message. Bloque jusqu'à ce qu'il y ait quelque chose à recevoir. Lève une *EOFError* s'il ne reste rien à recevoir et que l'autre côté de la connexion a été fermé.

buffer doit être un *bytes-like object* accessible en écriture. Si *offset* est donné, le message sera écrit dans le tampon à partir de cette position. *offset* doit être un entier positif, inférieur à la taille de *buffer* (en octets).

Si le tampon est trop petit une exception *BufferTooShort* est levée et le message complet est accessible via *e.args[0]* où *e* est l'instance de l'exception.

Modifié dans la version 3.3 : Les objets de connexions eux-mêmes peuvent maintenant être transférés entre les processus en utilisant *Connection.send()* et *Connection.recv()*.

Nouveau dans la version 3.3 : Les objets de connexions supportent maintenant le protocole des gestionnaires de contexte – voir *Le type gestionnaire de contexte*. *__enter__()* renvoie l'objet de connexion, et *__exit__()* appelle *close()*.

Par exemple :

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

Avertissement : La méthode `Connection.recv()` déserialise automatiquement les données qu'elle reçoit, ce qui peut être un risque de sécurité à moins que vous ne fassiez réellement confiance au processus émetteur du message.

Par conséquent, à moins que l'objet de connexion soit instancié par `Pipe()`, vous ne devriez uniquement utiliser les méthodes `recv()` et `send()` après avoir effectué une quelconque forme d'authentification. Voir *Clés d'authentification*.

Avertissement : Si un processus est tué pendant qu'il essaye de lire ou écrire sur le tube, alors les données du tube ont des chances d'être corrompues, parce qu'il devient impossible d'être sûr d'où se trouvent les bornes du message.

Primitives de synchronisation

Généralement les primitives de synchronisation ne sont pas nécessaire dans un programme multi-processus comme elles le sont dans un programme multi-fils d'exécution. Voir la documentation du module *threading*.

Notez que vous pouvez aussi créer des primitives de synchronisation en utilisant un objet gestionnaire – voir *Gestionnaires*.

class `multiprocessing.Barrier` (`parties`[, `action`[, `timeout`]])

Un objet barrière : un clone de *threading.Barrier*.

Nouveau dans la version 3.3.

class `multiprocessing.BoundedSemaphore` ([`value`])

Un objet sémaphore lié : un analogue proche de *threading.BoundedSemaphore*.

Une seule différence existe avec son proche analogue : le premier argument de sa méthode `acquire` est appelé *block*, pour la cohérence avec *Lock.acquire()*.

Note : Sur Mac OS X, elle n'est pas distinguable de la classe *Semaphore* parce que `sem_getvalue()` n'est pas implémentée sur cette plateforme.

class `multiprocessing.Condition` ([`lock`])

Une variable conditionnelle : un alias pour *threading.Condition*.

Si *lock* est spécifié il doit être un objet *Lock* ou *RLock* du module *multiprocessing*.

Modifié dans la version 3.3 : La méthode `wait_for()` a été ajoutée.

class `multiprocessing.Event`

Un clone de `threading.Event`.

class `multiprocessing.Lock`

Un verrou non récursif : un analogue proche de `threading.Lock`. Une fois que le processus ou le fil d'exécution a acquis un verrou, les tentatives suivantes d'acquisition depuis n'importe quel processus ou fil d'exécution bloqueront jusqu'à ce qu'il soit libéré ; n'importe quel processus ou fil peut le libérer. Les concepts et comportements de `threading.Lock` qui s'appliquent aux fils d'exécution sont répliqués ici dans `multiprocessing.Lock` et s'appliquent aux processus et aux fils d'exécution, à l'exception de ce qui est indiqué.

Notez que `Lock` est en fait une fonction *factory* qui renvoie une instance de `multiprocessing.synchronize.Lock` initialisée avec un contexte par défaut.

`Lock` supporte le protocole *context manager* et peut ainsi être utilisé avec une instruction `with`.

acquire (*block=True, timeout=None*)

Acquiert un verrou, bloquant ou non bloquant.

Avec l'argument *block* à `True` (par défaut), l'appel de méthode bloquera jusqu'à ce que le verrou soit dans déverrouillé, puis le verrouillera avant de renvoyer `True`. Notez que le nom de ce premier argument diffère de celui de `threading.Lock.acquire()`.

Avec l'argument *block* à `False`, l'appel de méthode ne bloque pas. Si le verrou est actuellement verrouillé, renvoie `False` ; autrement verrouille le verrou et renvoie `True`.

Quand invoqué avec un nombre flottant positif comme *timeout*, bloque au maximum pendant ce nombre spécifié de secondes, tant que le verrou ne peut être acquis. Les invocations avec une valeur de *timeout* négatives sont équivalents à zéro. Les invocations avec un *timeout* à `None` (par défaut) correspondent à un délai d'attente infini. Notez que le traitement des valeurs de *timeout* négatives et `None` diffère du comportement implémenté dans `threading.Lock.acquire()`. L'argument *timeout* n'a pas d'implication pratique si l'argument *block* est mis à `False` et est alors ignoré. Renvoie `True` si le verrou a été acquis et `False` si le temps de *timeout* a expiré.

release ()

Libère un verrou. Elle peut être appelée depuis n'importe quel processus ou fil d'exécution, pas uniquement le processus ou le fil qui a acquis le verrou à l'origine.

Le comportement est le même que `threading.Lock.release()` excepté que lorsque la méthode est appelée sur un verrou déverrouillé, une `ValueError` est levée.

class `multiprocessing.RLock`

Un objet verrou récursif : un analogue proche de `threading.RLock`. Un verrou récursif doit être libéré par le processus ou le fil d'exécution qui l'a acquis. Quand un processus ou un fil acquiert un verrou récursif, le même processus/fil peut l'acquérir à nouveau sans bloquer ; le processus/fil doit le libérer autant de fois qu'il l'a acquis.

Notez que `RLock` est en fait une fonction *factory* qui renvoie une instance de `multiprocessing.synchronize.RLock` initialisée avec un contexte par défaut.

`RLock` supporte le protocole *context manager* et peut ainsi être utilisée avec une instruction `with`.

acquire (*block=True, timeout=None*)

Acquiert un verrou, bloquant ou non bloquant.

Quand invoqué avec l'argument *block* à `True`, bloque jusqu'à ce que le verrou soit déverrouillé (n'appartenant à aucun processus ou fil d'exécution) sauf s'il appartient déjà au processus ou fil d'exécution courant. Le processus ou fil d'exécution courant prend la possession du verrou (s'il ne l'a pas déjà) et incrémente d'un le niveau de récursion du verrou, renvoyant ainsi `True`. Notez qu'il y a plusieurs différences dans le comportement de ce premier argument comparé à l'implémentation de `threading.RLock.acquire()`, à commencer par le nom de l'argument lui-même.

Quand invoqué avec l'argument *block* à `False`, ne bloque pas. Si le verrou est déjà acquis (et possédé) par un autre processus ou fil d'exécution, le processus/fil courant n'en prend pas la possession et le niveau de récursion n'est pas incrémenté, résultant en une valeur de retour à `False`. Si le verrou est déverrouillé, le processus/fil courant en prend possession et incrémente son niveau de récursion, renvoyant `True`.

L'usage et les comportements de l'argument *timeout* sont les mêmes que pour `Lock.acquire()`. Notez que certains de ces comportements diffèrent par rapport à ceux implémentés par `threading.RLock.acquire()`.

release()

Libère un verrou, décrémentant son niveau de récursion. Si après la décrémentation le niveau de récursion est zéro, réinitialise le verrou à un état déverrouillé (n'appartenant à aucun processus ou fil d'exécution) et si des processus/fils attendent que le verrou se déverrouille, autorise un seul d'entre-eux à continuer. Si après cette décrémentation le niveau de récursion est toujours strictement positif, le verrou reste verrouillé et propriété du processus/fil appelant.

N'appellez cette méthode que si le processus ou fil d'exécution appelant est propriétaire du verrou. Une `AssertionError` est levée si cette méthode est appelée par un processus/fil autre que le propriétaire ou si le verrou n'est pas verrouillé (possédé). Notez que le type d'exception levé dans cette situation diffère du comportement de `threading.RLock.release()`.

class multiprocessing.Semaphore([value])

Un objet sémaphore, proche analogue de `threading.Semaphore`.

Une seule différence existe avec son proche analogue : le premier argument de sa méthode `acquire` est appelé *block*, pour la cohérence avec `Lock.acquire()`.

Note : Sous Mac OS X, `sem_timedwait` n'est pas supporté, donc appeler `acquire()` avec un temps d'exécution limité émulerait le comportement de cette fonction en utilisant une boucle d'attente.

Note : Si le signal *SIGINT* généré par un Ctrl-C survient pendant que le fil d'exécution principal est bloqué par un appel à `BoundedSemaphore.acquire()`, `Lock.acquire()`, `RLock.acquire()`, `Semaphore.acquire()`, `Condition.acquire()` ou `Condition.wait()`, l'appel sera immédiatement interrompu et une `KeyboardInterrupt` sera levée.

Cela diffère du comportement de `threading` où le *SIGINT* est ignoré tant que les appels bloquants sont en cours.

Note : Certaines des fonctionnalités de ce paquet requièrent une implémentation fonctionnelle de sémaphores partagés sur le système hôte. Sans cela, le module `multiprocessing.synchronize` sera désactivé, et les tentatives de l'importer lèveront une `ImportError`. Voir [bpo-3770](#) pour plus d'informations.

Objets ctypes partagés

Il est possible de créer des objets partagés utilisant une mémoire partagée pouvant être héritée par les processus enfants.

multiprocessing.Value(typecode_or_type, *args, lock=True)

Renvoie un objet *ctypes* alloué depuis la mémoire partagée. Par défaut la valeur de retour est en fait un *wrapper* synchronisé autour de l'objet. L'objet en lui-même est accessible par l'attribut *value* de l'une *Value*.

typecode_or_type détermine le type de l'objet renvoyé : il s'agit soit d'un type *ctype* soit d'un caractère *typecode* tel qu'utilisé par le module `array`. **args* est passé au constructeur de ce type.

Si *lock* vaut `True` (par défaut), alors un nouveau verrou récursif est créé pour synchroniser l'accès à la valeur. Si *lock* est un objet `Lock` ou `RLock` alors il sera utilisé pour synchroniser l'accès à la valeur. Si *lock* vaut `False`, l'accès à l'objet renvoyé ne sera pas automatiquement protégé par un verrou, donc il ne sera pas forcément « *process-safe* ».

Les opérations telles que `+=` qui impliquent une lecture et une écriture ne sont pas atomiques. Ainsi si vous souhaitez par exemple réaliser une incrémentation atomique sur une valeur partagée, vous ne pouvez pas simplement faire

```
counter.value += 1
```

En supposant que le verrou associé est récursif (ce qui est le cas par défaut), vous pouvez à la place faire

```
with counter.get_lock():  
    counter.value += 1
```

Notez que *lock* est un argument *keyword-only*.

`multiprocessing.Array` (*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

Renvoie un tableau *ctypes* alloué depuis la mémoire partagée. Par défaut la valeur de retour est en fait un *wrapper* synchronisé autour du tableau.

typecode_or_type détermine le type des éléments du tableau renvoyé : il s'agit soit d'un type *ctype* soit d'un caractère *typecode* tel qu'utilisé par le module *array*. Si *size_or_initialize* est un entier, alors il détermine la taille du tableau, et le tableau sera initialisé avec des zéros. Autrement, *size*or_initializer* est une séquence qui sera utilisée pour initialiser le tableau et dont la taille détermine celle du tableau.

Si *lock* vaut *True* (par défaut), alors un nouveau verrou est créé pour synchroniser l'accès à la valeur. Si *lock* est un objet *Lock* ou *RLock* alors il sera utilisé pour synchroniser l'accès à la valeur. Si *lock* vaut *False*, l'accès à l'objet renvoyé ne sera pas automatiquement protégé par un verrou, donc il ne sera pas forcément « *process-safe* ».

Notez que *lock* est un argument *keyword-only*.

Notez qu'un tableau de *ctypes.c_char* a ses attributs *value* et *raw* qui permettent de l'utiliser pour stocker et récupérer des chaînes de caractères.

Le module `multiprocessing.sharedtypes`

Le module `multiprocessing.sharedtypes` fournit des fonctions pour allouer des objets *ctypes* depuis la mémoire partagée, qui peuvent être hérités par les processus fils.

Note : Bien qu'il soit possible de stocker un pointeur en mémoire partagée, rappelez-vous qu'un pointeur référence un emplacement dans l'espace d'adressage d'un processus particulier. Ainsi, ce pointeur a de fortes chances d'être invalide dans le contexte d'un autre processus et déréférencer le pointeur depuis ce second processus peut causer un plantage.

`multiprocessing.sharedtypes.RawArray` (*typecode_or_type*, *size_or_initializer*)

Renvoie un tableau *ctypes* alloué depuis la mémoire partagée.

typecode_or_type détermine le type des éléments du tableau renvoyé : il s'agit soit d'un type *ctype* soit d'un caractère codant le type des éléments du tableau (*typecode*) tel qu'utilisé par le module *array*. Si *size_or_initialize* est un entier, alors il détermine la taille du tableau, et le tableau sera initialisé avec des zéros. Autrement, *size_or_initializer* est une séquence qui sera utilisée pour initialiser le tableau et dont la taille détermine celle du tableau.

Notez que définir ou récupérer un élément est potentiellement non atomique – utilisez plutôt *Array()* pour vous assurer de synchroniser automatiquement avec un verrou.

`multiprocessing.sharedtypes.RawValue` (*typecode_or_type*, **args*)

Renvoie un objet *ctypes* alloué depuis la mémoire partagée.

typecode_or_type détermine le type de l'objet renvoyé : il s'agit soit d'un type *ctype* soit d'un caractère *typecode* tel qu'utilisé par le module *array*. **args* est passé au constructeur de ce type.

Notez que définir ou récupérer un élément est potentiellement non atomique – utilisez plutôt *Value()* pour vous assurer de synchroniser automatiquement avec un verrou.

Notez qu'un tableau de *ctypes.c_char* a ses attributs *value* et *raw* qui permettent de l'utiliser pour stocker et récupérer des chaînes de caractères – voir la documentation de *ctypes*.

`multiprocessing.sharedtypes.Array` (*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

Identique à *RawArray()* à l'exception que suivant la valeur de *lock* un *wrapper* de synchronisation *process-safe* pourra être renvoyé à la place d'un tableau *ctypes* brut.

Si *lock* vaut `True` (par défaut), alors un nouveau verrou est créé pour synchroniser l'accès à la valeur. Si *lock* est un objet `Lock` ou `RLock` alors il sera utilisé pour synchroniser l'accès à la valeur. Si *lock* vaut `False`, l'accès à l'objet renvoyé ne sera pas automatiquement protégé par un verrou, donc il ne sera pas forcément « *process-safe* ». Notez que *lock* est un argument *keyword-only*.

`multiprocessing.sharedctypes.Value` (*typecode_or_type*, **args*, *lock=True*)

Identique à `RawValue()` à l'exception que suivant la valeur de *lock* un *wrapper* de synchronisation *process-safe* pourra être renvoyé à la place d'un objet *ctypes* brut.

Si *lock* vaut `True` (par défaut), alors un nouveau verrou est créé pour synchroniser l'accès à la valeur. Si *lock* est un objet `Lock` ou `RLock` alors il sera utilisé pour synchroniser l'accès à la valeur. Si *lock* vaut `False`, l'accès à l'objet renvoyé ne sera pas automatiquement protégé par un verrou, donc il ne sera pas forcément « *process-safe* ». Notez que *lock* est un argument *keyword-only*.

`multiprocessing.sharedctypes.copy` (*obj*)

Renvoie un objet *ctypes* alloué depuis la mémoire partagée, qui est une copie de l'objet *ctypes* *obj*.

`multiprocessing.sharedctypes.synchronized` (*obj*[, *lock*])

Renvoie un *wrapper* *process-safe* autour de l'objet *ctypes* qui utilise *lock* pour synchroniser l'accès. Si *lock* est `None` (par défaut), un objet `multiprocessing.RLock` est créé automatiquement.

Un *wrapper* synchronisé aura deux méthodes en plus de celles de l'objet qu'il enveloppe : `get_obj()` renvoie l'objet *wrapped* et `get_lock()` renvoie le verrou utilisé pour la synchronisation.

Notez qu'accéder à l'objet *ctypes* à travers le *wrapper* peut s'avérer beaucoup plus lent qu'accéder directement à l'objet *ctypes* brut.

Modifié dans la version 3.5 : Les objets synchronisés supportent le protocole *context manager*.

Le tableau ci-dessous compare la syntaxe de création des objets *ctypes* partagés depuis une mémoire partagée avec la syntaxe normale *ctypes*. (Dans le tableau, `MyStruct` est une sous-classe quelconque de `ctypes.Structure`.)

<i>ctypes</i>	<i>sharedctypes</i> utilisant un type	<i>sharedctypes</i> utilisant un <i>typecode</i>
<code>c_double(2.4)</code>	<code>RawValue(c_double, 2.4)</code>	<code>RawValue("d", 2.4)</code>
<code>MyStruct(4, 6)</code>	<code>RawValue(MyStruct, 4, 6)</code>	
<code>(c_short * 7)()</code>	<code>RawArray(c_short, 7)</code>	<code>RawArray("h", 7)</code>
<code>(c_int * 3)(9, 2, 8)</code>	<code>RawArray(c_int, (9, 2, 8))</code>	<code>RawArray("i", (9, 2, 8))</code>

Ci-dessous un exemple où des objets *ctypes* sont modifiés par un processus fils :

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value *= 2
    x.value *= 2
    s.value = s.value.upper()
    for a in A:
        a.x *= 2
        a.y *= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
```

(suite sur la page suivante)

(suite de la page précédente)

```

s = Array('c', b'hello world', lock=lock)
A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

p = Process(target=modify, args=(n, x, s, A))
p.start()
p.join()

print(n.value)
print(x.value)
print(s.value)
print([(a.x, a.y) for a in A])

```

Les résultats affichés sont

```

49
0.11111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]

```

Gestionnaires

Les gestionnaires fournissent un moyen de créer des données qui peuvent être partagées entre les différents processus, incluant le partage à travers le réseau entre des processus tournant sur des machines différentes. Un objet gestionnaire contrôle un processus serveur qui gère les *shared objects*. Les autres processus peuvent accéder aux objets partagés à l'aide de mandataires.

`multiprocessing.Manager()`

Renvoie un objet *SyncManager* démarré qui peut être utilisé pour partager des objets entre les processus. L'objet gestionnaire renvoyé correspond à un processus enfant instancié et possède des méthodes pour créer des objets partagés et renvoyer les mandataires correspondants.

Les processus gestionnaires seront arrêtés dès qu'ils seront collectés par le ramasse-miettes ou que leur processus parent se terminera. Les classes gestionnaires sont définies dans le module `multiprocessing.managers` :

class `multiprocessing.managers.BaseManager` (`[address[, authkey]]`)

Crée un objet *BaseManager*.

Une fois créé il faut appeler `start()` ou `get_server().serve_forever()` pour assurer que l'objet gestionnaire référence un processus gestionnaire démarré.

address est l'adresse sur laquelle le processus gestionnaire écoute pour de nouvelles connexions. Si *address* est `None`, une adresse arbitraire est choisie.

authkey est la clé d'authentification qui sera utilisée pour vérifier la validité des connexions entrantes sur le processus serveur. Si *authkey* est `None` alors `current_process().authkey` est utilisée. Autrement *authkey* est utilisée et doit être une chaîne d'octets.

start (`[initializer[, initargs]]`)

Démarré un sous-processus pour démarrer le gestionnaire. Si *initializer* n'est pas `None` alors le sous-processus appellera `initializer(*initargs)` quand il démarrera.

get_server ()

Renvoie un objet *Server* qui représente le serveur sous le contrôle du gestionnaire. L'objet *Server* supporte la méthode `serve_forever()` :

```

>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=(' ', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()

```


Server possède en plus un attribut *address*.

connect ()

Connecte un objet gestionnaire local au processus gestionnaire distant :

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 5000), authkey=b'abc')
>>> m.connect()
```

shutdown ()

Stoppe le processus utilisé par le gestionnaire. Cela est disponible uniquement si *start ()* a été utilisée pour démarrer le processus serveur.

Cette méthode peut être appelée plusieurs fois.

register (typeid[, callable[, proxytype[, exposed[, method_to_typeid[, create_method]]]]])

Une méthode de classe qui peut être utilisée pour enregistrer un type ou un callable avec la classe gestionnaire.

typeid est un « *type identifier* » qui est utilisé pour identifier un type particulier d'objet partagé. Cela doit être une chaîne de caractères.

callable est un objet callable utilisé pour créer les objets avec cet identifiant de type. Si une instance de gestionnaire prévoit de se connecter au serveur en utilisant sa méthode *connect ()* ou si l'argument *create_method* vaut *False* alors cet argument peut être laissé à *None*.

proxytype est une sous-classe de *BaseProxy* utilisée pour créer des mandataires autour des objets partagés avec ce *typeid*. S'il est *None*, une classe mandataire sera créée automatiquement.

exposed est utilisé pour préciser une séquence de noms de méthodes dont les mandataires pour ce *typeid* doivent être autorisés à accéder via *BaseProxy._callmethod()*. (Si *exposed* est *None* alors *proxytype._exposed_* est utilisé à la place s'il existe.) Dans le cas où aucune liste *exposed* n'est précisée, toutes les « méthodes publiques » de l'objet partagé seront accessibles. (Ici une « méthode publique » signifie n'importe quel attribut qui possède une méthode *__call__()* et dont le nom ne commence pas par un *'_'*.)

method_to_typeid est un tableau associatif utilisé pour préciser le type de retour de ces méthodes exposées qui doivent renvoyer un mandataire. Il associe un nom de méthode à une chaîne de caractères *typeid*. (Si *method_to_typeid* est *None*, *proxytype._method_to_typeid_* est utilisé à la place s'il existe). Si le nom d'une méthode n'est pas une clé de ce tableau associatif ou si la valeur associée est *None*, l'objet renvoyé par la méthode sera une copie de la valeur.

create_method détermine si une méthode devrait être créée avec le nom *typeid*, qui peut être utilisée pour indiquer au processus serveur de créer un nouvel objet partagé et d'en renvoyer un mandataire. a valeur par défaut est *True*.

Les instances de *BaseManager* ont aussi une propriété en lecture seule :

address

L'adresse utilisée par le gestionnaire.

Modifié dans la version 3.3 : Les objets gestionnaires supportent le protocole des gestionnaires de contexte – voir *Le type gestionnaire de contexte*. *__enter__()* démarre le processus serveur (s'il n'a pas déjà été démarré) et renvoie l'objet gestionnaire. *__exit__()* appelle *shutdown()*.

Dans les versions précédentes *__enter__()* ne démarrerait pas le processus serveur du gestionnaire s'il n'était pas déjà démarré.

class multiprocessing.managers.SyncManager

Une sous-classe de *BaseManager* qui peut être utilisée pour la synchronisation entre processus. Des objets de ce type sont renvoyés par *multiprocessing.Manager()*.

Ces méthodes créent et renvoient des *Objets mandataires* pour un certain nombre de types de données communément utilisés pour être synchronisés entre les processus. Elles incluent notamment des listes et dictionnaires partagés.

Barrier (parties[, action[, timeout]])

Crée un objet *threading.Barrier* partagé et renvoie un mandataire pour cet objet.

Nouveau dans la version 3.3.

BoundedSemaphore ([*value*])

Crée un objet `threading.BoundedSemaphore` partagé et renvoie un mandataire pour cet objet.

Condition ([*lock*])

Crée un objet `threading.Condition` partagé et renvoie un mandataire pour cet objet.

Si *lock* est fourni alors il doit être un mandataire pour un objet `threading.Lock` ou `threading.RLock`.

Modifié dans la version 3.3 : La méthode `wait_for()` a été ajoutée.

Event ()

Crée un objet `threading.Event` partagé et renvoie un mandataire pour cet objet.

Lock ()

Crée un objet `threading.Lock` partagé et renvoie un mandataire pour cet objet.

Namespace ()

Crée un objet `Namespace` partagé et renvoie un mandataire pour cet objet.

Queue ([*maxsize*])

Crée un objet `queue.Queue` partagé et renvoie un mandataire pour cet objet.

RLock ()

Crée un objet `threading.RLock` partagé et renvoie un mandataire pour cet objet.

Semaphore ([*value*])

Crée un objet `threading.Semaphore` partagé et renvoie un mandataire pour cet objet.

Array (*typecode*, *sequence*)

Crée un tableau et renvoie un mandataire pour cet objet.

Value (*typecode*, *value*)

Crée un objet avec un attribut *value* accessible en écriture et renvoie un mandataire pour cet objet.

dict ()

dict (*mapping*)

dict (*sequence*)

Crée un objet `dict` partagé et renvoie un mandataire pour cet objet.

list ()

list (*sequence*)

Crée un objet `list` partagé et renvoie un mandataire pour cet objet.

Modifié dans la version 3.6 : Les objets partagés peuvent être imbriqués. Par exemple, un conteneur partagé tel qu'une liste partagée peu contenir d'autres objets partagés qui seront aussi gérés et synchronisés par le `SyncManager`.

class multiprocessing.managers.**Namespace**

Un type qui peut être enregistré avec `SyncManager`.

Un espace de nommage n'a pas de méthodes publiques, mais possède des attributs accessibles en écriture. Sa représentation montre les valeurs de ses attributs.

Cependant, en utilisant un mandataire pour un espace de nommage, un attribut débutant par '_' est un attribut du mandataire et non de l'objet cible :

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3      # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

Gestionnaires personnalisés

Pour créer son propre gestionnaire, il faut créer une sous-classe de `BaseManager` et utiliser la méthode de classe `register()` pour enregistrer de nouveaux types ou *callable*s au gestionnaire. Par exemple :

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))          # prints 7
        print(maths.mul(7, 8))         # prints 56
```

Utiliser un gestionnaire distant

Il est possible de lancer un serveur gestionnaire sur une machine et d'avoir des clients l'utilisant sur d'autres machines (en supposant que les pare-feus impliqués l'autorisent).

Exécuter les commandes suivantes crée un serveur pour une simple queue partagée à laquelle des clients distants peuvent accéder :

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Un client peut accéder au serveur comme suit :

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

Un autre client peut aussi l'utiliser :

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

Les processus locaux peuvent aussi accéder à cette queue, utilisant le code précédent sur le client pour y accéder à distance :

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super(Worker, self).__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=(' ', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Objets mandataires

Un mandataire est un objet qui *réfère* un objet partagé appartenant (supposément) à un processus différent. L'objet partagé est appelé le *référé* du mandataire. Plusieurs mandataires peuvent avoir un même référé.

Un mandataire possède des méthodes qui appellent les méthodes correspondantes du référé (bien que toutes les méthodes du référé ne soient pas nécessairement accessibles à travers le mandataire). De cette manière, un mandataire peut être utilisé comme le serait son référé :

```
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

Notez qu'appliquer `str()` à un mandataire renvoie la représentation du référé, alors que `repr()` renvoie celle du mandataire.

Une fonctionnalité importante des objets mandataires est qu'ils sont sérialisables et peuvent donc être échangés entre les processus. Ainsi, un référé peut contenir des *Objets mandataires*. Cela permet d'imbriquer des listes et dictionnaires gérés ainsi que d'autres *Objets mandataires* :

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)           # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

De même, les mandataires de listes et dictionnaires peuvent être imbriqués dans d'autres :

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

Si des objets standards (non *proxyfiés*) *list* ou *dict* sont contenus dans un référent, les modifications sur ces valeurs mutables ne seront pas propagées à travers le gestionnaire parce que le mandataire n'a aucun moyen de savoir quand les valeurs contenues sont modifiées. Cependant, stocker une valeur dans un conteneur mandataire (qui déclenche un appel à `__setitem__` sur le mandataire) propage bien la modification à travers le gestionnaire et modifie effectivement l'élément, il est ainsi possible de réassigner la valeur modifiée au conteneur mandataire :

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

Cette approche est peut-être moins pratique que d'utiliser des *Objets mandataires* imbriqués pour la majorité des cas d'utilisation, mais démontre aussi un certain niveau de contrôle sur la synchronisation.

Note : Les types de mandataires de *multiprocessing* n'implémentent rien pour la comparaison par valeurs. Par exemple, on a :

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

Il faut à la place simplement utiliser une copie du référent pour faire les comparaisons.

class multiprocessing.managers.BaseProxy

Les objets mandataires sont des instances de sous-classes de *BaseProxy*.

__callmethod (*methodname* [, *args* [, *kws*]])

Appelle et renvoie le résultat d'une méthode du référent du mandataire.

Si *proxy* est un mandataire sont le référent est *obj*, alors l'expression

```
proxy._callmethod(methodname, args, kwds)
```

s'évalue comme

```
getattr(obj, methodname)(*args, **kwds)
```

dans le processus du gestionnaire.

La valeur renvoyée sera une copie du résultat de l'appel ou un mandataire sur un nouvel objet partagé – voir l'a documentation de l'argument *method_to_typeid* de *BaseManager.register()*.

Si une exception est levée par l'appel, elle est relayée par *_callmethod()*. Si une autre exception est levée par le processus du gestionnaire, elle est convertie en une *RemoteError* et est levée par *_callmethod()*. Notez en particulier qu'une exception est levée si *methodname* n'est pas *exposée*.

Un exemple d'utilisation de *_callmethod()* :

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()

Renvoie une copie du référent.

Si le référent n'est pas sérialisable, une exception est levée.

__repr__()

Renvoie la représentation de l'objet mandataire.

__str__()

Renvoie la représentation du référent.

Nettoyage

Un mandataire utilise un *callback* sous une référence faible de façon à ce que quand il est collecté par le ramasse-miettes, il se désenregistre auprès du gestionnaire qui possède le référent.

Un objet partagé est supprimé par le processus gestionnaire quand plus aucun mandataire ne le référence.

Bassins de processus

On peut créer un bassin de processus qui exécuteront les tâches qui lui seront soumises avec la classe *Pool*.

```
class multiprocessing.pool.Pool([processes[, initializer[, initargs[, maxtasksperchild[, context]]]])
```

Un objet *process pool* qui contrôle un bassin de processus *workers* auquel sont soumises des tâches. Il supporte les résultats asynchrones avec des *timeouts* et des *callbacks* et possède une implémentation parallèle de *map*.

processes est le nombre de processus *workers* à utiliser. Si *processes* est *None*, le nombre renvoyé par *os.cpu_count()* est utilisé.

Si *initializer* n'est pas *None*, chaque processus *worker* appellera *initializer(*initargs)* en démarrant.

maxtasksperchild est le nombre de tâches qu'un processus *worker* peut accomplir avant de se fermer et d'être remplacé par un *worker* frais, pour permettre aux ressources inutilisées d'être libérées. Par défaut *maxtasksperchild* est *None*, ce qui signifie que le *worker* vit aussi longtemps que le bassin.

context peut être utilisé pour préciser le contexte utilisé pour démarrer les processus *workers*. Habituellement un bassin est créé à l'aide de la fonction `multiprocessing.Pool()` ou de la méthode `Pool()` d'un objet de contexte. Dans les deux cas *context* est réglé de façon appropriée.

Notez que les méthodes de l'objet *pool* ne doivent être appelées que par le processus qui l'a créé.

Nouveau dans la version 3.2 : *maxtasksperchild*

Nouveau dans la version 3.4 : *context*

Note : Les processus *workers* à l'intérieur d'une *Pool* vivent par défaut aussi longtemps que la queue de travail du bassin. Un modèle fréquent chez d'autres systèmes (tels qu'Apache, *mod_wsgi*, etc.) pour libérer les ressources détenues par les *workers* est d'autoriser un *worker* dans le bassin à accomplir seulement une certaine charge de travail avant de se fermer, se retrouvant nettoyé et remplacé par un nouveau processus fraîchement lancé. L'argument *maxtasksperchild* de *Pool* expose cette fonctionnalité à l'utilisateur final.

apply (*func*[, *args*[, *kwargs*]])

Appelle *func* avec les arguments *args* et les arguments nommés *kwargs*. Bloque jusqu'à ce que le résultat soit prêt. En raison de ce blocage, `apply_async()` est préférable pour exécuter du travail en parallèle. De plus, *func* est exécutée sur un seul des *workers* du bassin.

apply_async (*func*[, *args*[, *kwargs*[, *callback*[, *error_callback*]]]])

Une variante de la méthode `apply()` qui renvoie un objet résultat.

Si *callback* est précisé alors ce doit être un objet callable qui accepte un seul argument. Quand le résultat est prêt, *callback* est appelé avec ce résultat, si l'appel n'échoue pas auquel cas *error_callback* est appelé à la place.

Si *error_callback* est précisé alors ce doit être un objet callable qui accepte un seul argument. Si la fonction cible échoue, alors *error_callback* est appelé avec l'instance de l'exception.

Les *callbacks* doivent se terminer immédiatement, autrement le fil d'exécution qui gère les résultats se retrouverait bloqué.

map (*func*, *iterable*[, *chunksize*])

Un équivalent parallèle à la fonction *built-in* `map()` (qui ne supporte cependant qu'un *itérable* en argument). Elle bloque jusqu'à ce que le résultat soit prêt.

La méthode découpe l'itérable en un nombre de morceaux qu'elle envoie au bassin de processus comme des tâches séparées. La taille (approximative) de ces morceaux peut être précisée en passant à *chunksize* un entier positif.

map_async (*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

Une variante de la méthode `map()` qui renvoie un objet résultat.

Si *callback* est précisé alors ce doit être un objet callable qui accepte un seul argument. Quand le résultat est prêt, *callback* est appelé avec ce résultat, si l'appel n'échoue pas auquel cas *error_callback* est appelé à la place.

Si *error_callback* est précisé alors ce doit être un objet callable qui accepte un seul argument. Si la fonction cible échoue, alors *error_callback* est appelé avec l'instance de l'exception.

Les *callbacks* doivent se terminer immédiatement, autrement le fil d'exécution qui gère les résultats se retrouverait bloqué.

imap (*func*, *iterable*[, *chunksize*])

A la version de `map()`.

L'argument *chunksize* est le même que celui utilisé par la méthode `map()`. Pour de très longs itérables, utiliser une grande valeur pour *chunksize* peut faire s'exécuter la tâche **beaucoup** plus rapidement qu'en utilisant la valeur par défaut de 1.

Aussi, si *chunksize* vaut 1 alors la méthode `next()` de l'itérateur renvoyé par `imap()` prend un paramètre optionnel *timeout* : `next(timeout)` lève une `multiprocessing.TimeoutError` si le résultat ne peut pas être renvoyé avant *timeout* secondes.

imap_unordered (*func*, *iterable*[, *chunksize*])

Identique à `imap()` si ce n'est que l'ordre des résultats de l'itérateur renvoyé doit être considéré comme arbitraire. (L'ordre n'est garanti que quand il n'y a qu'un *worker*.)

starmap (*func*, *iterable*_[, *chunksize*])

Semblable à `map()` à l'exception que les éléments d'*iterable* doivent être des itérables qui seront dépaquetés comme arguments pour la fonction.

Par conséquent un *iterable* `[(1, 2), (3, 4)]` donnera pour résultat `[func(1, 2), func(3, 4)]`.

Nouveau dans la version 3.3.

starmap_async (*func*, *iterable*_{[, *chunksize*_{[, *callback*_[, *error_callback*]]]})}

Une combinaison de `starmap()` et `map_async()` qui itère sur *iterable* (composé d'itérables) et appelle *func* pour chaque itérable dépaqueté. Renvoie l'objet résultat.

Nouveau dans la version 3.3.

close ()

Empêche de nouvelles tâches d'être envoyées à la *pool*. Les processus *workers* se terminent une fois que toutes les tâches ont été complétées.

terminate ()

Stoppe immédiatement les processus *workers* sans finaliser les travaux courants. Quand l'objet *pool* est collecté par le ramasse-miettes, sa méthode `terminate()` est appelée immédiatement.

join ()

Attend que les processus *workers* se terminent. Il est nécessaire d'appeler `close()` ou `terminate()` avant d'utiliser `join()`.

Nouveau dans la version 3.3 : Les bassins de *workers* supportent maintenant le protocole des gestionnaires de contexte – voir *Le type gestionnaire de contexte*. `__enter__()` renvoie l'objet *pool* et `__exit__()` appelle `terminate()`.

class multiprocessing.pool.AsyncResult

La classe des résultats renvoyés par `Pool.apply_async()` et `Pool.map_async()`.

get ([*timeout*])

Renvoie le résultat quand il arrive. Si *timeout* n'est pas `None` et que le résultat n'arrive pas avant *timeout* secondes, une `multiprocessing.TimeoutError` est levée. Si l'appel distance lève une exception, alors elle est relayée par `get()`.

wait ([*timeout*])

Attend que le résultat soit disponible ou que *timeout* secondes s'écoulent.

ready ()

Renvoie `True` ou `False` suivant si la tâche est accomplie.

successful ()

Renvoie `True` ou `False` suivant si la tâche est accomplie sans lever d'exception. Lève une `AssertionError` si le résultat n'est pas prêt.

Les exemples suivants présentent l'utilisation d'un bassin de *workers* :

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a
        ↪ single process
        print(result.get(timeout=1))        # prints "100" unless your computer is
        ↪ *very* slow

        print(pool.map(f, range(10)))      # prints "[0, 1, 4, ..., 81]"

        it = pool.imap(f, range(10))
```

(suite sur la page suivante)

(suite de la page précédente)

```

print(next(it))          # prints "0"
print(next(it))          # prints "1"
print(it.next(timeout=1)) # prints "4" unless your computer is_
↪ *very* slow

result = pool.apply_async(time.sleep, (10,))
print(result.get(timeout=1)) # raises multiprocessing.TimeoutError

```

Auditeurs et Clients

Habituellement l'échange de messages entre processus est réalisé en utilisant des queues ou des objets *Connection* renvoyés par *Pipe()*.

Cependant, le module *multiprocessing.connection* permet un peu plus de flexibilité. Il fournit un message de plus haut-niveau orienté API pour gérer des connecteurs ou des tubes nommés sous Windows. Il gère aussi l'authentification par condensat (*digest authentication* en anglais) en utilisant le module *hmac*, et pour interroger de multiples connexions en même temps.

multiprocessing.connection.deliver_challenge(*connection*, *authkey*)

Envoie un message généré aléatoirement à l'autre bout de la connexion et attend une réponse.

Si la réponse correspond au condensat du message avec la clé *authkey*, alors un message de bienvenue est envoyé à l'autre bout de la connexion. Autrement, une *AuthenticationError* est levée.

multiprocessing.connection.answer_challenge(*connection*, *authkey*)

Reçoit un message, calcule le condensat du message en utilisant la clé *authkey*, et envoie le condensat en réponse.

Si un message de bienvenue n'est pas reçu, une *AuthenticationError* est levée.

multiprocessing.connection.Client(*address*[, *family*[, *authkey*]])

Essaie d'établir une connexion avec l'auditeur qui utilise l'adresse *address*, renvoie une *Connection*.

Le type de la connexion est déterminé par l'argument *family*, mais il peut généralement être omis puisqu'il peut être inféré depuis le format d'*address*. (Voir *Formats d'adresses*)

Si *authkey* est passée et n'est pas *None*, elle doit être une chaîne d'octets et sera utilisée comme clé secrète pour le défi d'authentification basé sur HMAC. Aucune authentification n'est réalisée si *authkey* est *None*. Une *AuthenticationError* est levée si l'authentification échoue. Voir *Clés d'authentification*.

class multiprocessing.connection.Listener([*address*[, *family*[, *backlog*[, *authkey*]]]])

Une enveloppe autour d'un connecteur lié ou un tube nommé sous Windows qui écoute pour des connexions.

address est l'adresse à utiliser par le connecteur lié ou le tube nommé de l'objet auditeur.

Note : Si une adresse "0.0.0.0" est utilisée, l'adresse ne sera pas un point d'accès connectable sous Windows. Si vous avez besoin d'un point d'accès connectable, utilisez "127.0.0.1".

family est le type de connecteur (ou tube nommé) à utiliser. Cela peut être l'une des chaînes 'AF_INET' (pour un connecteur TCP), 'AF_UNIX' (pour un connecteur Unix) ou 'AF_PIPE' (pour un tube nommé sous Windows). Seulement le premier d'entre eux est garanti d'être disponible. Si *family* est *None*, la famille est inférée depuis le format d'*address*. Si *address* est aussi *None*, la famille par défaut est utilisée. La famille par défaut est supposée être la plus rapide disponible. Voir *Formats d'adresses*. Notez que si la *family* est 'AF_UNIX' et qu'*address* est *None*, le connecteur est créé dans un répertoire temporaire privé créé avec *tempfile.mkstemp()*.

Si l'objet auditeur utilise un connecteur alors *backlog* (1 par défaut) est passé à la méthode *listen()* du connecteur une fois qu'il a été lié.

Si *authkey* est passée et n'est pas *None*, elle doit être une chaîne d'octets et sera utilisée comme clé secrète pour le défi d'authentification basé sur HMAC. Aucune authentification n'est réalisée si *authkey* est *None*. Une *AuthenticationError* est levée si l'authentification échoue. Voir *Clés d'authentification*.

accept ()

Accepte une connexion sur le connecteur lié ou le tube nommé de l'objet auditeur et renvoie un objet *Connection*. Si la tentative d'authentification échoue, une *AuthenticationError* est levée.

close ()

Ferme le connecteur lié ou le tube nommé de l'objet auditeur. La méthode est appelée automatiquement quand l'auditeur est collecté par le ramasse-miettes. Il est cependant conseillé de l'appeler explicitement.

Les objets auditeurs ont aussi les propriétés en lecture seule suivantes :

address

L'adresse utilisée par l'objet auditeur.

last_accepted

L'adresse depuis laquelle a été établie la dernière connexion. *None* si aucune n'est disponible.

Nouveau dans la version 3.3 : Les objets auditeurs supportent maintenant le protocole des gestionnaires de contexte – voir *Le type gestionnaire de contexte*. `__enter__()` renvoie l'objet auditeur, et `__exit__()` appelle `close()`.

`multiprocessing.connection.wait (object_list, timeout=None)`

Attend qu'un objet d'*object_list* soit prêt. Renvoie la liste de ces objets d'*object_list* qui sont prêts. Si *timeout* est un nombre flottant, l'appel bloquera au maximum ce nombre de secondes. Si *timeout* est *None*, l'appel bloquera pour une durée non limitée. Un *timeout* négatif est équivalent à un *timeout* nul.

Pour Unix et Windows, un objet peut apparaître dans *object_list* s'il est

- un objet *Connection* accessible en lecture ;
- un objet *socket.socket* connecté et accessible en lecture ; ou
- l'attribut *sentinel* d'un objet *Process*.

Une connexion (*socket* en anglais) est prête quand il y a des données disponibles en lecture dessus, ou que l'autre bout a été fermé.

Unix : `wait(object_list, timeout)` est en grande partie équivalente à `select.select(object_list, [], [], timeout)`. La différence est que, si `select.select()` est interrompue par un signal, elle peut lever une *OSError* avec un numéro d'erreur *EINTR*, alors que `wait()` ne le fera pas.

Windows : un élément d'*object_list* doit être soit un identifiant *waitable* (en accord avec la définition utilisée par la documentation de la fonction `Win32 WaitForMultipleObjects()`), soit un objet avec une méthode `fileno()` qui renvoie un identifiant de connecteur ou de tube (notez que les identifiants de tubes et de connecteurs **ne sont pas** des identifiants *waitables*).

Nouveau dans la version 3.3.

Exemples

Le code serveur suivant crée un auditeur qui utilise 'secret password' comme clé d'authentification. Il attend ensuite une connexion et envoie les données au client :

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)      # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

Le code suivant se connecte au serveur et en reçoit des données :

```

from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())           # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())     # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr)) # => 8
    print(arr)                   # => array('i', [42, 1729, 0, 0, 0])

```

Le code suivant utilise `wait()` pour attendre des messages depuis plusieurs processus à la fois :

```

import time, random
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it. This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)

```

Formats d'adresses

- Une adresse 'AF_INET' est un *tuple* de la forme (hostname, port) où *hostname* est une chaîne et *port* un entier.
- Une adresse 'AF_UNIX' est une chaîne représentant un nom de fichier sur le système de fichiers.
- Une adresse 'AF_PIPE' est une chaîne de la forme `r'\.\pipe{PipeName}'`. Pour utiliser un `Client()` pour se connecter à un tube nommé sur une machine distante appelée *ServerName*, il faut plutôt utiliser une adresse de la forme `r'\ServerName\pipe{PipeName}'`.

Notez que toute chaîne commençant par deux antislashes est considérée par défaut comme l'adresse d'un 'AF_PIPE' plutôt qu'une adresse 'AF_UNIX'.

Clés d'authentification

Quand `Connection.recv` est utilisée, les données reçues sont automatiquement désérialisées par *pickle*. Malheureusement désérialiser des données depuis une source non sûre constitue un risque de sécurité. Par conséquent `Listener` et `Client()` utilisent le module *hmac* pour fournir une authentification par condensat.

Une clé d'authentification est une chaîne d'octets qui peut être vue comme un mot de passe : quand une connexion est établie, les deux interlocuteurs vont demander à l'autre une preuve qu'il connaît la clé d'authentification. (Démontrer que les deux utilisent la même clé n'implique **pas** d'échanger la clé sur la connexion.)

Si l'authentification est requise et qu'aucune clé n'est spécifiée alors la valeur de retour de `current_process().authkey` est utilisée (voir *Process*). Cette valeur est automatiquement héritée par tout objet *Process* créé par le processus courant. Cela signifie que (par défaut) tous les processus d'un programme multi-processus partageront une clé d'authentification unique qui peut être utilisée pour mettre en place des connexions entre-eux.

Des clés d'authentification adaptées peuvent aussi être générées par `os.urandom()`.

Journalisation

Un certain support de la journalisation est disponible. Notez cependant que le paquet *logging* n'utilise pas de verrous partagés entre les processus et il est donc possible (dépendant du type de gestionnaire) que les messages de différents processus soient mélangés.

`multiprocessing.get_logger()`

Renvoie le journaliseur utilisé par *multiprocessing*. Si nécessaire, un nouveau sera créé.

À sa première création le journaliseur a pour niveau `logging.NOTSET` et pas de gestionnaire par défaut. Les messages envoyés à ce journaliseur ne seront pas propagés par défaut au journaliseur principal.

Notez que sous Windows les processus fils n'hériteront que du niveau du journaliseur du processus parent – toute autre personnalisation du journaliseur ne sera pas héritée.

`multiprocessing.log_to_stderr()`

Cette fonction effectue un appel à `get_logger()` mais en plus de renvoyer le journaliseur créé par `get_logger`, elle ajoute un gestionnaire qui envoie la sortie sur `sys.stderr` en utilisant le format `'[(levelname)s/ % (processName)s] % (message)s'`.

L'exemple ci-dessous présente une session avec la journalisation activée :

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
```

(suite sur la page suivante)

(suite de la page précédente)

```
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pypm-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

Pour un tableau complet des niveaux de journalisation, voir le module *logging*.

Le module `multiprocessing.dummy`

multiprocessing.dummy réplique toute l'API de *multiprocessing* mais n'est rien de plus qu'un *wrapper* autour du module *threading*.

17.2.3 Lignes directrices de programmation

Il y a certaines lignes directrices et idiomes auxquels il faut adhérer en utilisant *multiprocessing*.

Toutes les méthodes de démarrage

Les règles suivantes s'appliquent aux méthodes de démarrage.

Éviter les états partagés

Autant que possible, vous devriez éviter de déplacer de larges données entre les processus.

Il est probablement meilleur de s'en tenir à l'utilisation de queues et tubes pour la communication entre processus plutôt que d'utiliser des primitives de synchronisation plus bas-niveau.

Sérialisation

Assurez-vous que les arguments passés aux méthodes des mandataires soient sérialisables (*pickables*).

Sûreté des mandataires à travers les fils d'exécution

N'utilisez pas d'objet mandataire depuis plus d'un fil d'exécution à moins que vous ne le protégiez avec un verrou.

(Il n'y a jamais de problème avec plusieurs processus utilisant un *même* mandataire.)

Attendre les processus zombies

Sous Unix quand un processus se termine mais n'est pas attendu, il devient un zombie. Il ne devrait jamais y en avoir beaucoup parce que chaque fois qu'un nouveau processus démarre (ou que *active_children()* est appelée) tous les processus complétés qui n'ont pas été attendus le seront. Aussi appeler la méthode *Process.is_alive* d'un processus terminé attendra le processus. Toutefois il est probablement une bonne pratique d'attendre explicitement tous les processus que vous démarrez.

Préférez hériter que sérialiser/désérialiser

Quand vous utilisez les méthodes de démarrage *spawn* ou *forkserver*, de nombreux types de *multiprocessing* nécessitent d'être sérialisés pour que les processus enfants puissent les utiliser. Cependant, il faut généralement éviter d'envoyer des objets partagés aux autres processus en utilisant des tubes ou des queues. Vous devriez plutôt vous arranger pour qu'un processus qui nécessite l'accès à une ressource partagée crée autre part qu'il en hérite depuis un de ses processus ancêtres.

Éviter de terminer les processus

Utiliser la méthode `Process.terminate` pour stopper un processus risque de casser ou de rendre indisponible aux autres processus des ressources partagées (comme des verrous, sémaphores, tubes et queues) actuellement utilisée par le processus.

Il est donc probablement préférable de n'utiliser `Process.terminate` que sur les processus qui n'utilisent jamais de ressources partagées.

Attendre les processus qui utilisent des queues

Gardez à l'esprit qu'un processus qui a placé des éléments dans une queue attendra que tous les éléments mis en tampon soient consommés par le fil d'exécution « consommateur » du tube sous-jacent avant de se terminer. (Le processus enfant peut appeler la méthode `Queue.cancel_join_thread` de la queue pour éviter ce comportement.)

Cela signifie que chaque fois que vous utilisez une queue vous devez vous assurer que tous les éléments qui y ont été placés seront effectivement supprimés avant que le processus ne soit attendu. Autrement vous ne pouvez pas être sûr que les processus qui ont placé des éléments dans la queue se termineront. Souvenez-vous aussi que tous les processus non *daemons* seront attendus automatiquement.

L'exemple suivant provoquera un interblocage :

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                # this deadlocks
    obj = queue.get()
```

Une solution ici serait d'intervertir les deux dernières lignes (ou simplement supprimer la ligne `p.join()`).

Passer explicitement les ressources aux processus fils

Sous Unix en utilisant la méthode de démarrage *fork*, un processus fils peut utiliser une ressource partagée créée par un processus parent en utilisant une ressource globale. Cependant, il est préférable de passer l'objet en argument au constructeur du processus fils.

En plus de rendre le code (potentiellement) compatible avec Windows et les autres méthodes de démarrage, cela assure aussi que tant que le processus fils est en vie, l'objet ne sera pas collecté par le ramasse-miettes du processus parent. Cela peut être important si certaines ressources sont libérées quand l'objet est collecté par le ramasse-miettes du processus parent.

Donc par exemple

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

devrait être réécrit comme

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...
```

(suite sur la page suivante)

(suite de la page précédente)

```
if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Faire attention à remplacer `sys.stdin` par un objet « *file-like* »

À l'origine, `multiprocessing` appelait inconditionnellement :

```
os.close(sys.stdin.fileno())
```

dans la méthode `multiprocessing.Process._bootstrap()` — cela provoquait des problèmes avec les processus imbriqués. Cela peut être changé en :

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

Qui résout le problème fondamental des collisions entre processus provoquant des erreurs de mauvais descripteurs de fichiers, mais introduit un potentiel danger pour les applications qui remplacent `sys.stdin()` avec un « *file-like object* » ayant une sortie *bufferisée*. Ce danger est que si plusieurs processus appellent `close()` sur cet objet *file-like*, cela peut amener les données à être transmises à l'objet à plusieurs reprises, résultant en une corruption.

Si vous écrivez un objet *file-like* et implémentez votre propre cache, vous pouvez le rendre sûr pour les *forks* en stockant le *pid* chaque fois que vous ajoutez des données au cache, et annulez le cache quand le *pid* change. Par exemple :

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

Pour plus d'informations, voir [bpo-5155](#), [bpo-5313](#) et [bpo-5331](#)

Les méthodes de démarrage *spawn* et *forkserver*

Certaines restrictions ne s'appliquent pas à la méthode de démarrage *fork*.

Plus de sérialisation

Assurez-vous que tous les arguments de `Process.__init__()` sont sérialisables avec *pickle*. Aussi, si vous héritez de `Process`, assurez-vous que toutes les instances sont sérialisables quand la méthode `Process.start` est appelée.

Variables globales

Gardez en tête que si le code exécuté dans un processus fils essaie d'accéder à une variable globale, alors la valeur qu'il voit (s'il y en a une) pourrait ne pas être la même que la valeur du processus parent au moment même où `Process.start` est appelée.

Cependant, les variables globales qui sont juste des constantes de modules ne posent pas de problèmes.

Importation sûre du module principal

Assurez-vous que le module principal peut être importé en toute sécurité par un nouvel interpréteur Python sans causer d'effets de bord inattendus (comme le démarrage d'un nouveau processus).

Par exemple, utiliser la méthode de démarrage *spawn* ou *forkserver* pour lancer le module suivant échouerait avec une `RuntimeError` :

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

Vous devriez plutôt protéger le « point d'entrée » du programme en utilisant `if __name__ == '__main__':` comme suit :

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

(La ligne `freeze_support()` peut être omise si le programme est uniquement lancé normalement et pas gelé.)

Cela permet aux interpréteurs Python fraîchement instanciés d'importer en toute sécurité le module et d'exécution ensuite la fonction `foo()`.

Des restrictions similaires s'appliquent si une *pool* ou un gestionnaire est créé dans le module principal.

17.2.4 Exemples

Démonstration de comment créer et utiliser des gestionnaires et mandataires personnalisés :

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
```

(suite sur la page suivante)

(suite de la page précédente)

```

def __next__(self):
    return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))

```

(suite sur la page suivante)

(suite de la page précédente)

```
print('op.pow(2, 94) =', op.pow(2, 94))
print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

En utilisant *Pool* :

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
    )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
```

(suite sur la page suivante)

(suite de la page précédente)

```

# Tests
#

TASKS = [(mul, (i, 7)) for i in range(10)] + \
        [(plus, (i, 8)) for i in range(10)]

results = [pool.apply_async(calculate, t) for t in TASKS]
imap_it = pool.imap(calculatestar, TASKS)
imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

print('Ordered results using pool.apply_async():')
for r in results:
    print('\t', r.get())
print()

print('Ordered results using pool.imap():')
for x in imap_it:
    print('\t', x)
print()

print('Unordered results using pool.imap_unordered():')
for x in imap_unordered_it:
    print('\t', x)
print()

print('Ordered results using pool.map() --- will block till complete:')
for x in pool.map(calculatestar, TASKS):
    print('\t', x)
print()

#
# Test error handling
#

print('Testing error handling:')

try:
    print(pool.apply(f, (5,)))
except ZeroDivisionError:
    print('\tGot ZeroDivisionError as expected from pool.apply()')
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print(pool.map(f, list(range(10))))
except ZeroDivisionError:
    print('\tGot ZeroDivisionError as expected from pool.map()')
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print(list(pool.imap(f, list(range(10)))))
except ZeroDivisionError:
    print('\tGot ZeroDivisionError as expected from list(pool.imap())')
else:
    raise AssertionError('expected ZeroDivisionError')

```

(suite sur la page suivante)

(suite de la page précédente)

```

it = pool.imap(f, list(range(10)))
for i in range(10):
    try:
        x = next(it)
    except ZeroDivisionError:
        if i == 5:
            pass
    except StopIteration:
        break
    else:
        if i == 5:
            raise AssertionError('expected ZeroDivisionError')

assert i == 9
print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
print()

#
# Testing timeouts
#

print('Testing ApplyResult.get() with timeout:', end=' ')
res = pool.apply_async(calculate, TASKS[0])
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('\n\t%s' % res.get(0.02))
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')
print()
print()

print('Testing IMapIterator.next() with timeout:', end=' ')
it = pool.imap(calculatestar, TASKS)
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('\n\t%s' % it.next(0.02))
    except StopIteration:
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')
print()
print()

if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()

```

Un exemple montrant comment utiliser des queues pour alimenter en tâches une collection de processus *workers* et collecter les résultats :

```

import time
import random

```

(suite sur la page suivante)

(suite de la page précédente)

```

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):

```

(suite sur la page suivante)

(suite de la page précédente)

```
print('\t', done_queue.get())

# Add more tasks using `put()`
for task in TASKS2:
    task_queue.put(task)

# Get and print some more results
for i in range(len(TASKS2)):
    print('\t', done_queue.get())

# Tell child processes to stop
for i in range(NUMBER_OF_PROCESSES):
    task_queue.put('STOP')

if __name__ == '__main__':
    freeze_support()
    test()
```

17.3 Le paquet concurrent

Il n'y a actuellement qu'un module dans ce paquet :

— `concurrent.futures` – Lancer des tâches en parallèle

17.4 `concurrent.futures` — Launching parallel tasks

Nouveau dans la version 3.2.

Source code : `Lib/concurrent/futures/thread.py` and `Lib/concurrent/futures/process.py`

The `concurrent.futures` module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using `ThreadPoolExecutor`, or separate processes, using `ProcessPoolExecutor`. Both implement the same interface, which is defined by the abstract `Executor` class.

17.4.1 Executor Objects

class `concurrent.futures.Executor`

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

submit (*fn*, *args, **kwargs)

Schedules the callable, *fn*, to be executed as *fn*(*args **kwargs) and returns a `Future` object representing the execution of the callable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map (*func*, **iterables*, *timeout=None*, *chunksize=1*)

Similar to `map(func, *iterables)` except :

- the *iterables* are collected immediately rather than lazily;
- *func* is executed asynchronously and several calls to *func* may be made concurrently.

The returned iterator raises a `concurrent.futures.TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to `Executor.map()`. *timeout* can be an int or a float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If a *func* call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

When using `ProcessPoolExecutor`, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With `ThreadPoolExecutor`, *chunksize* has no effect.

Modifié dans la version 3.5 : Added the *chunksize* argument.

shutdown (*wait=True*)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to `Executor.submit()` and `Executor.map()` made after shutdown will raise `RuntimeError`.

If *wait* is `True` then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If *wait* is `False` then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of *wait*, the entire Python program will not exit until all pending futures are done executing.

You can avoid having to call this method explicitly if you use the `with` statement, which will shutdown the `Executor` (waiting as if `Executor.shutdown()` were called with *wait* set to `True`):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

17.4.2 ThreadPoolExecutor

`ThreadPoolExecutor` is an `Executor` subclass that uses a pool of threads to execute calls asynchronously.

Deadlocks can occur when the callable associated with a `Future` waits on the results of another `Future`. For example :

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result()) # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result()) # a will never complete because it is waiting on b.
    return 6
```

(suite sur la page suivante)

(suite de la page précédente)

```

executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)

```

Et :

```

def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)

```

class `concurrent.futures.ThreadPoolExecutor` (*max_workers=None, thread_name_prefix=""*)

An *Executor* subclass that uses a pool of at most *max_workers* threads to execute calls asynchronously.

Modifié dans la version 3.5 : If *max_workers* is *None* or not given, it will default to the number of processors on the machine, multiplied by 5, assuming that *ThreadPoolExecutor* is often used to overlap I/O instead of CPU work and the number of workers should be higher than the number of workers for *ProcessPoolExecutor*.

Nouveau dans la version 3.6 : The *thread_name_prefix* argument was added to allow users to control the *threading.Thread* names for worker threads created by the pool for easier debugging.

ThreadPoolExecutor Example

```

import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://some-made-up-domain.com/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))

```


17.4.3 ProcessPoolExecutor

The `ProcessPoolExecutor` class is an `Executor` subclass that uses a pool of processes to execute calls asynchronously. `ProcessPoolExecutor` uses the `multiprocessing` module, which allows it to side-step the *Global Interpreter Lock* but also means that only picklable objects can be executed and returned.

The `__main__` module must be importable by worker subprocesses. This means that `ProcessPoolExecutor` will not work in the interactive interpreter.

Calling `Executor` or `Future` methods from a callable submitted to a `ProcessPoolExecutor` will result in deadlock.

class `concurrent.futures.ProcessPoolExecutor` (*max_workers=None*)

An `Executor` subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If *max_workers* is `None` or not given, it will default to the number of processors on the machine. If *max_workers* is lower or equal to 0, then a `ValueError` will be raised.

Modifié dans la version 3.3 : When one of the worker processes terminates abruptly, a `BrokenProcessPool` error is now raised. Previously, behaviour was undefined but operations on the executor or its futures would often freeze or deadlock.

ProcessPoolExecutor Example

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

17.4.4 Future Objects

The *Future* class encapsulates the asynchronous execution of a callable. *Future* instances are created by *Executor.submit()*.

class `concurrent.futures.Future`

Encapsulates the asynchronous execution of a callable. *Future* instances are created by *Executor.submit()* and should not be created directly except for testing.

cancel()

Attempt to cancel the call. If the call is currently being executed and cannot be cancelled then the method will return `False`, otherwise the call will be cancelled and the method will return `True`.

cancelled()

Return `True` if the call was successfully cancelled.

running()

Return `True` if the call is currently being executed and cannot be cancelled.

done()

Return `True` if the call was successfully cancelled or finished running.

result (*timeout=None*)

Return the value returned by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a `concurrent.futures.TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call raised, this method will raise the same exception.

exception (*timeout=None*)

Return the exception raised by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a `concurrent.futures.TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call completed without raising, `None` is returned.

add_done_callback (*fn*)

Attaches the callable *fn* to the future. *fn* will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an `Exception` subclass, it will be logged and ignored. If the callable raises a `BaseException` subclass, the behavior is undefined.

If the future has already completed or been cancelled, *fn* will be called immediately.

The following *Future* methods are meant for use in unit tests and *Executor* implementations.

set_running_or_notify_cancel()

This method should only be called by *Executor* implementations before executing the work associated with the *Future* and by unit tests.

If the method returns `False` then the *Future* was cancelled, i.e. *Future.cancel()* was called and returned `True`. Any threads waiting on the *Future* completing (i.e. through *as_completed()* or *wait()*) will be woken up.

If the method returns `True` then the *Future* was not cancelled and has been put in the running state, i.e. calls to *Future.running()* will return `True`.

This method can only be called once and cannot be called after *Future.set_result()* or *Future.set_exception()* have been called.

set_result (*result*)

Sets the result of the work associated with the *Future* to *result*.

This method should only be used by *Executor* implementations and unit tests.

set_exception (*exception*)

Sets the result of the work associated with the *Future* to the *Exception* exception.

This method should only be used by *Executor* implementations and unit tests.

17.4.5 Module Functions

`concurrent.futures.wait` (*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

Wait for the *Future* instances (possibly created by different *Executor* instances) given by *fs* to complete. Returns a named 2-tuple of sets. The first set, named *done*, contains the futures that completed (finished or were cancelled) before the wait completed. The second set, named *not_done*, contains uncompleted futures.

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or *None*, there is no limit to the wait time.

return_when indique quand la fonction doit se terminer. Il peut prendre les valeurs suivantes :

Constante	Description
FIRST_COMPLETED	La fonction se termine lorsque n'importe quel futur se termine ou est annulé.
FIRST_EXCEPTION	La fonction se termine lorsque n'importe quel futur se termine en levant une exception. Si aucun <i>futur</i> ne lève d'exception, équivaut à <i>ALL_COMPLETED</i> .
ALL_COMPLETED	La fonction se termine lorsque les <i>futurs</i> sont tous finis ou annulés.

`concurrent.futures.as_completed` (*fs*, *timeout=None*)

Returns an iterator over the *Future* instances (possibly created by different *Executor* instances) given by *fs* that yields futures as they complete (finished or were cancelled). Any futures given by *fs* that are duplicated will be returned once. Any futures that completed before *as_completed()* is called will be yielded first. The returned iterator raises a `concurrent.futures.TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to *as_completed()*. *timeout* can be an int or float. If *timeout* is not specified or *None*, there is no limit to the wait time.

Voir aussi :

PEP 3148 – futures - execute computations asynchronously The proposal which described this feature for inclusion in the Python standard library.

17.4.6 Exception classes

exception `concurrent.futures.CancelledError`

Raised when a future is cancelled.

exception `concurrent.futures.TimeoutError`

Raised when a future operation exceeds the given timeout.

exception `concurrent.futures.process.BrokenProcessPool`

Derived from *RuntimeError*, this exception class is raised when one of the workers of a *ProcessPoolExecutor* has terminated in a non-clean fashion (for example, if it was killed from the outside).

Nouveau dans la version 3.3.

17.5 subprocess — Gestion de sous-processus

Code source : [Lib/subprocess.py](#)

Le module `subprocess` vous permet de lancer de nouveaux processus, les connecter à des tubes d'entrée/sortie/erreur, et d'obtenir leurs codes de retour. Ce module a l'intention de remplacer plusieurs anciens modules et fonctions :

```
os.system
os.spawn*
```

De plus amples informations sur comment le module `subprocess` peut être utilisé pour remplacer ces modules et fonctions peuvent être trouvées dans les sections suivantes.

Voir aussi :

PEP 324 – PEP proposant le module `subprocess`

17.5.1 Utiliser le module `subprocess`

L'approche recommandée pour invoquer un sous-processus et d'utiliser la fonction `run()` pour tous les cas d'utilisation qu'il gère. Pour les cas d'utilisation plus avancés, l'interface inhérente `Popen` peut être utilisée directement.

La fonction `run()` a été ajoutée avec Python 3.5 ; si vous avez besoin d'une compatibilité avec des versions plus anciennes, référez-vous à la section [Ancienne interface \(API\) haut-niveau](#).

`subprocess.run(args, *, stdin=None, input=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None, check=False, encoding=None, errors=None, env=None)`

Lance la commande décrite par `args`. Attend que la commande se termine, puis renvoie une instance `CompletedProcess`.

Les arguments présentés ci-dessus sont simplement les plus utilisés, décrits ci-dessous dans la section [Arguments fréquemment utilisés](#) (d'où l'utilisation de la notation *keyword-only* dans la signature abrégée). La signature complète de la fonction est sensiblement la même que celle du constructeur de `Popen` - à l'exception de `timeout`, `input` et `check`, tous les arguments donnés à cette fonction passent à travers cette interface.

Les sorties standard (`stdout`) et d'erreur (`stderr`) ne sont pas capturées par défaut. Pour les capturer, utilisez `PIPE` comme valeur des arguments `stdout` et/ou `stderr`.

L'argument `timeout` est passé à `Popen.communicate()`. Si le `timeout` expire, le processus enfant sera tué et attendu. Une exception `TimeoutExpired` sera levée une fois que le processus enfant se sera terminé.

L'argument `input` est passé à `Popen.communicate()` et donc à l'entrée standard (`stdin`) du sous-processus. Si l'argument est utilisé, il doit contenir une séquence de *bytes*, ou une chaîne de caractères si `encoding` ou `errors` sont spécifiés, ou si `universal_newlines` est vrai. Quand utilisé, l'objet interne `Popen` est automatiquement créé avec `stdin=PIPE`, et l'argument `stdin` ne doit donc pas être utilisé.

Si `check` est vrai, et que le processus s'arrête avec un code de statut non nul, une exception `CalledProcessError` est levée. Les attributs de cette exception contiennent les arguments, le code de statut, et les sorties standard et d'erreur si elles ont été capturées.

Si `encoding` ou `errors` sont spécifiés, ou `universal_newlines` est vrai, les fichiers pour les entrées et sorties sont ouverts en mode texte en utilisant les paramètres `encoding` et `errors` spécifiés, ou les valeurs par défaut de `io.TextIOWrapper`. Autrement, les fichiers sont ouverts en mode binaire.

Si `env` n'est pas `None`, il doit être un tableau associatif définissant les variables d'environnement du nouveau processus ; elles seront utilisées à la place du comportement par défaut qui est d'hériter de l'environnement du processus courant. Il est passé directement à `Popen`.

Exemples :

```

>>> subprocess.run(["ls", "-l"]) # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], stdout=subprocess.PIPE)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n')

```

Nouveau dans la version 3.5.

Modifié dans la version 3.6 : Ajout des paramètres *encoding* et *errors*

class subprocess.CompletedProcess

La valeur de retour de `run()`, représentant un processus qui s'est terminé.

args

Les arguments utilisés pour lancer le processus. Cela peut être une liste ou une chaîne de caractères.

returncode

Le code de statut du processus fils. Typiquement, un code de statut de 0 indique qu'il s'est exécuté avec succès. Une valeur négative $-N$ indique que le processus enfant a été terminé par un signal N (seulement sur les systèmes *POSIX*).

stdout

La sortie standard capturée du processus enfant. Une séquence de *bytes*, ou une chaîne de caractères si `run()` a été appelée avec *encoding* ou *errors*. Vaut `None` si la sortie standard n'était pas capturée.

Si vous avez lancé le processus avec `stderr=subprocess.STDOUT`, les sorties standard et d'erreur seront combinées dans cet attribut, et `stderr` sera mis à `None`.

stderr

La sortie d'erreur capturée du processus enfant. Une séquence de *bytes*, ou une chaîne de caractères si `run()` a été appelée avec *encoding* ou *errors*. Vaut `None` si la sortie d'erreur n'était pas capturée.

check_returncode()

Si *returncode* n'est pas nul, lève une *CalledProcessError*.

Nouveau dans la version 3.5.

subprocess.DEVNULL

Valeur spéciale qui peut être utilisée pour les arguments *stdin*, *stdout* ou *stderr* de *Popen* et qui indique que le fichier spécial *os.devnull* sera utilisé.

Nouveau dans la version 3.3.

subprocess.PIPE

Valeur spéciale qui peut être utilisée pour les arguments *stdin*, *stdout* ou *stderr* de *Popen* et qui indique qu'un tube vers le flux standard doit être ouvert. Surtout utile avec *Popen.communicate()*.

subprocess.STDOUT

Valeur spéciale qui peut être utilisée pour l'argument *stderr* de *Popen* et qui indique que la sortie d'erreur doit être redirigée vers le même descripteur que la sortie standard.

exception subprocess.SubprocessError

Classe de base à toutes les autres exceptions du module.

Nouveau dans la version 3.3.

exception subprocess.TimeoutExpired

Sous-classe de *SubprocessError*, levée quand un *timeout* expire pendant l'attente d'un processus enfant.

cmd

La commande utilisée pour instancier le processus fils.

timeout

Le *timeout* en secondes.

output

La sortie du processus fils, si capturée par `run()` ou `check_output()`. Autrement, `None`.

stdout

Alias pour *output*, afin d'avoir une symétrie avec *stderr*.

stderr

La sortie d'erreur du processus fils, si capturée par `run()`. Autrement, `None`.

Nouveau dans la version 3.3.

Modifié dans la version 3.5 : Ajout des attributs *stdout* et *stderr*

exception subprocess.CalledProcessError

Sous-classe de *SubprocessError*, levée quand un processus lancé par `check_call()` ou `check_output()` renvoie un code de statut non nul.

returncode

Code de statut du processus fils. Si le processus a été arrêté par un signal, le code sera négatif et correspondra à l'opposé du numéro de signal.

cmd

La commande utilisée pour instancier le processus fils.

output

La sortie du processus fils, si capturée par `run()` ou `check_output()`. Autrement, `None`.

stdout

Alias pour *output*, afin d'avoir une symétrie avec *stderr*.

stderr

La sortie d'erreur du processus fils, si capturée par `run()`. Autrement, `None`.

Modifié dans la version 3.5 : Ajout des attributs *stdout* et *stderr*

Arguments fréquemment utilisés

Pour gérer un large ensemble de cas, le constructeur de *Popen* (et les fonctions de convenance) acceptent de nombreux arguments optionnels. Pour les cas d'utilisation les plus typiques, beaucoup de ces arguments peuvent sans problème être laissés à leurs valeurs par défaut. Les arguments les plus communément nécessaires sont :

args est requis pour tous les appels et doit être une chaîne de caractères ou une séquence d'arguments du programme. Il est généralement préférable de fournir une séquence d'arguments, puisque cela permet au module de s'occuper des potentiels échappements ou guillemets autour des arguments (p. ex. pour permettre des espaces dans des noms de fichiers). Si l'argument est passé comme une simple chaîne, soit *shell* doit valoir *True* (voir ci-dessous) soit la chaîne doit simplement contenir le nom du programme à exécuter sans spécifier d'arguments supplémentaires.

stdin, *stdout* et *stderr* spécifient respectivement les descripteurs d'entrée standard, de sortie standard et de sortie d'erreur du programme exécuté. Les valeurs acceptées sont *PIPE*, *DEVNULL*, un descripteur de fichier existant (nombre entier positif), un objet de fichier, et `None`. *PIPE* indique qu'un nouveau tube vers le processus enfant sera créé. *DEVNULL* indique que le fichier spécial *os.devnull* sera utilisé. Avec les paramètres `None` par défaut, aucune redirection ne se produira, les descripteurs de fichiers du fils seront hérités du parent. Additionnellement, *stderr* peut valoir *STDOUT*, qui indique que les données de la sortie d'erreur du processus fils doivent être capturées dans le même descripteur de fichier que la sortie standard.

Si *encoding* ou *errors* sont spécifiés, ou si *universal_newlines* est vrai, les fichiers *stdin*, *stdout* et *stderr* seront ouverts en mode texte en utilisant les *encoding* et *errors* spécifiés à l'appel, ou les valeurs par défaut de *io.TextIOWrapper*.

Pour *stdin*, les caractères de fin de ligne `'\n'` de l'entrée seront convertis vers des séparateurs de ligne par défaut *os.linesep*. Pour *stdout* et *stderr*, toutes les fins de lignes des sorties seront converties vers `'\n'`. Pour plus d'informations, voir la documentation de la classe *io.TextIOWrapper* quand l'argument *newline* du constructeur est `None`.

Si le mode texte n'est pas utilisé, *stdin*, *stdout* et *stderr* seront ouverts comme des flux binaires. Aucune conversion d'encodage ou de fins de ligne ne sera réalisée.

Nouveau dans la version 3.6 : Ajout des paramètres *encoding* et *errors*.

Note : L'attribut *newlines* des objets *Popen.stdin*, *Popen.stdout* et *Popen.stderr* ne sont pas mis à jour par la méthode *Popen.communicate()*.

Si *shell* vaut *True*, la commande spécifiée sera exécutée à travers un *shell*. Cela peut être utile si vous utilisez Python pour le contrôle de flot qu'il propose par rapport à beaucoup de *shells* système et voulez tout de même profiter des autres fonctionnalités des *shells* telles que les tubes (*pipes*), les motifs de fichiers, l'expansion des variables d'environnement, et l'expansion du *~* vers le répertoire d'accueil de l'utilisateur. Cependant, notez que Python lui-même propose l'implémentation de beaucoup de ces fonctionnalités (en particulier *glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()* et *shutil*).

Modifié dans la version 3.3 : Quand *universal_newlines* vaut *True*, la classe utilise l'encodage *locale.getpreferredencoding(False)* plutôt que *locale.getpreferredencoding()*. Voir la classe *io.TextIOWrapper* pour plus d'informations sur ce changement.

Note : Lire la section *Security Considerations* avant d'utiliser *shell=True*.

Ces options, ainsi que toutes les autres, sont décrites plus en détails dans la documentation du constructeur de *Popen*.

Constructeur de *Popen*

La création et la gestion sous-jacentes des processus est gérée par la classe *Popen*. Elle offre beaucoup de flexibilité de façon à ce que les développeurs soient capables de gérer les cas d'utilisation les moins communs, non couverts par les fonctions de convenance.

```
class subprocess.Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None,
                        preexec_fn=None, close_fds=True, shell=False, cwd=None, env=None, univer-
                        sal_newlines=False, startupinfo=None, creationflags=0, restore_signals=True,
                        start_new_session=False, pass_fds=(), *, encoding=None, errors=None)
```

Exécute un programme fils dans un nouveau processus. Sur les systèmes *POSIX*, la classe utilise un comportement similaire à *os.execvp()* pour exécuter le programme. Sur Windows, la classe utilise la fonction Windows *CreateProcess()*. Les arguments de *Popen* sont les suivants.

args doit être une séquence d'arguments du programme ou une chaîne seule. Par défaut, le programme à exécuter est le premier élément de *args* si *args* est une séquence. Si *args* est une chaîne, l'interprétation dépend de la plateforme et est décrite plus bas. Voir les arguments *shell* et *executable* pour d'autres différences avec le comportement par défaut. Sans autre indication, il est recommandé de passer *args* comme une séquence.

Sur les systèmes *POSIX*, si *args* est une chaîne, elle est interprétée comme le nom ou le chemin du programme à exécuter. Cependant, cela ne peut être fait que si le programme est passé sans arguments.

Note : *shlex.split()* peut être utilisée pour déterminer le découpage correct de *args*, spécifiquement dans les cas complexes :

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo '
↪$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```


Notez en particulier que les options (comme *-input*) et arguments (comme *eggs.txt*) qui sont séparés par des espaces dans le *shell* iront dans des éléments séparés de la liste, alors que les arguments qui nécessitent des guillemets et échappements quand utilisés dans le *shell* (comme les noms de fichiers contenant des espaces ou la commande *echo* montrée plus haut) forment des éléments uniques.

Sous Windows, si *args* est une séquence, elle sera convertie vers une chaîne de caractères de la manière décrite dans [Convertir une séquence d'arguments vers une chaîne de caractères sous Windows](#). Cela fonctionne ainsi parce que la fonction `CreateProcess()` opère sur des chaînes.

L'argument *shell* (qui vaut `False` par défaut) spécifie s'il faut utiliser un *shell* comme programme à exécuter. Si *shell* vaut `True`, il est recommandé de passer *args* comme une chaîne de caractères plutôt qu'une séquence.

Sur les systèmes POSIX avec *shell=True*, le *shell* par défaut est `/bin/sh`. Si *args* est une chaîne de caractères, la chaîne spécifie la commande à exécuter à travers le *shell*. Cela signifie que la chaîne doit être formatée exactement comme elle le serait si elle était tapée dans l'invite de commandes du *shell*. Cela inclut, par exemple, les guillemets ou les *backslashes* échappant les noms de fichiers contenant des espaces. Si *args* est une séquence, le premier élément spécifie la commande, et les éléments supplémentaires seront traités comme des arguments additionnels à passer au *shell* lui-même. Pour ainsi dire, *Popen* réalise l'équivalent de :

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

Sous Windows avec *shell=True*, la variable d'environnement `COMSPEC` spécifie le *shell* par défaut. La seule raison pour laquelle vous devriez spécifier *shell=True* sous Windows est quand la commande que vous souhaitez exécuter est une *built-in* du *shell* (p. ex. **dir** ou **copy**). Vous n'avez pas besoin de *shell=True* pour lancer un fichier batch ou un exécutable console.

Note : Lire la section [Security Considerations](#) avant d'utiliser *shell=True*.

bufsize sera fourni comme l'argument correspondant à la fonction `open()`, lors de la création des objets de fichiers pour les tubes *stdin/stdout/stderr* :

- 0 indique de ne pas utiliser de tampon (les lectures et écritures sont des appels systèmes et peuvent renvoyer des données incomplètes) ;
- 1 indique une mise en cache par ligne (utilisable seulement si `universal_newlines=True`, c'est-à-dire en mode texte) ;
- toutes les autres valeurs positives indiquent d'utiliser un tampon d'approximativement cette taille ;
- un *bufsize* négatif (par défaut) indique au système d'utiliser la valeur par défaut `io.DEFAULT_BUFFER_SIZE`.

Modifié dans la version 3.3.1 : *bufsize* vaut maintenant `-1` par défaut, pour activer par défaut la mise en cache et correspondre au comportement attendu par la plupart des codes. Dans les versions de Python antérieures à 3.2.4 et 3.3.1, par erreur, la valeur par défaut était 0 qui ne réalisait aucune mise en cache et autorisait les lectures incomplètes. Cela n'était pas intentionnel et ne correspondait pas au comportement de Python 2 attendu par la plupart des codes.

L'argument *executable* spécifie un programme de remplacement à exécuter. Il est très rarement nécessaire. Quand *shell=False*, *executable* remplace le programme à exécuter spécifié par *args*. Cependant, les arguments originaux d'*args* sont toujours passés au programme. La plupart des programmes traitent le programme spécifié par **args* comme le nom de la commande, qui peut être différent du programme réellement exécuté. Sur les systèmes POSIX, le nom tiré d'*args* devient le nom affiché pour l'exécutable dans des utilitaires tels que **ps**. Si *shell=True*, sur les systèmes POSIX, l'argument *executable* précise le *shell* à utiliser plutôt que `/bin/sh` par défaut.

stdin, *stdout* et *stderr* spécifient respectivement les descripteurs d'entrée standard, de sortie standard et de sortie d'erreur du programme exécuté. Les valeurs acceptées sont *PIPE*, *DEVNULL*, un descripteur de fichier existant (nombre entier positif), un *file object*, et `None`. *PIPE* indique qu'un nouveau tube vers le processus enfant sera créé. *DEVNULL* indique que le fichier spécial `os.devnull` sera utilisé. Avec les paramètres `None` par défaut, aucune redirection ne se produira, les descripteurs de fichiers du fils seront hérités du parent. Additionnellement, *stderr* peut valoir *STDOUT*, qui indique que les données de la sortie d'erreur du processus fils doivent être capturées dans le même descripteur de fichier que la sortie standard.

Si un objet callable est passé à *preexec_fn*, cet objet sera appelé dans le processus enfant juste avant d'exécuter le programme. (POSIX seulement)

Avertissement : Le paramètre `preexec_fn` n'est pas sain à utiliser en présence d'autres fils d'exécution dans votre application. Le processus fils pourrait être bloqué (*deadlock*) avant qu'`exec` ne soit appelée. Si vous devez utiliser ce paramètre, gardez son utilisation triviale ! Minimisez le nombre de bibliothèques que vous y appelez.

Note : Si vous devez modifier l'environnement du fils, utilisez le paramètre `env` plutôt que faire cela dans une `preexec_fn`. Le paramètre `start_new_session` peut prendre la place de `preexec_fn` qui était autrefois communément utilisé pour appeler `os.setsid()` dans le fils.

Si `close_fds` est vrai, tous les descripteurs de fichiers à l'exception de 0, 1 et 2 seront fermés avant que le processus fils ne soit exécuté. (POSIX seulement). La valeur par défaut varie selon les plateformes : il est toujours vrai sur les systèmes POSIX. Sous Windows, il est vrai quand `stdin/stdout/stderr` sont `None`, faux autrement. Sous Windows, si `close_fds` est vrai, aucun gestionnaire ne sera hérité par le processus fils. Notez que sous Windows, vous ne pouvez pas mettre `close_fds` à `true` et en même temps rediriger les entrées/sorties standards avec les paramètres `stdin/stdout/stderr`.

Modifié dans la version 3.2 : La valeur par défaut de `close_fds` n'est plus `False`, comme décrit ci-dessus.

`pass_fds` est une séquence optionnelle de descripteurs de fichiers à garder ouverts entre le parent et l'enfant. Fournir `pass_fds` force `close_fds` à valoir `True`. (POSIX seulement)

Nouveau dans la version 3.2 : Ajout du paramètre `pass_fds`.

Si `cwd` n'est pas `None`, la fonction change de répertoire courant pour `cwd` avant d'exécuter le fils. `cwd` peut être un objet `str` ou un *chemin-compatible*. En particulier, la fonction recherche *executable* (ou le premier élément d'`args`) relativement à `cwd` si le chemin d'exécution est relatif.

Modifié dans la version 3.6 : le paramètre `cwd` accepte un *path-like object*.

Si `restore_signals` est vrai (par défaut), tous les signaux que Python a mis à `SIG_IGN` sont restaurés à `SIG_DFL` dans le processus fils avant l'appel à `exec`. Actuellement, cela inclut les signaux `SIGPIPE`, `SIGXFSZ` et `SIGXFSZ`. (POSIX seulement)

Modifié dans la version 3.2 : Ajout de `restore_signals`.

Si `start_new_session` est vrai, l'appel système à `setsid()` sera réalisé dans le processus fils avant l'exécution du sous-processus. (POSIX seulement)

Modifié dans la version 3.2 : Ajout de `start_new_session`.

Si `env` n'est pas `None`, il doit être un tableau associatif définissant les variables d'environnement du nouveau processus ; elles seront utilisées à la place du comportement par défaut qui est d'hériter de l'environnement du processus courant.

Note : Si spécifié, `env` doit fournir chaque variable requise pour l'exécution du programme. Sous Windows, afin d'exécuter un *side-by-side assembly*, l'environnement `env` spécifié **doit** contenir une variable `SystemRoot` valide.

Si `encoding` ou `errors` sont spécifiés, les fichiers `stdin`, `stdout` et `stderr` sont ouverts en mode texte, avec l'encodage et la valeur d'`errors` spécifiés, comme décrit ci-dessus dans *Arguments fréquemment utilisés*. Si `universal_newlines` vaut `True`, ils sont ouverts en mode texte avec l'encodage par défaut. Autrement, ils sont ouverts comme des flux binaires.

Nouveau dans la version 3.6 : Ajout d'`encoding` et `errors`.

Si fourni, `startupinfo` sera un objet `STARTUPINFO`, qui sera passé à la fonction `CreateProcess` inhérente. `creationflags`, si fourni, peut valoir `CREATE_NEW_CONSOLE` ou `CREATE_NEW_PROCESS_GROUP`. (Windows seulement)

Les objets `Popen` sont gérés comme gestionnaires de contexte avec l'instruction `with` : à la sortie, les descripteurs de fichiers standards sont fermés, et le processus est attendu

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Modifié dans la version 3.2 : Ajout de la gestion des gestionnaires de contexte.

Modifié dans la version 3.6 : Le destructeur de *Popen* émet maintenant un avertissement *ResourceWarning* si le processus fils est toujours en cours d'exécution.

Exceptions

Les exceptions levées dans le processus fils, avant que le nouveau programme n'ait commencé son exécution, seront relayées dans le parent. Additionnellement, l'objet de l'exception aura un attribut supplémentaire appelé *child_traceback*, une chaîne de caractères contenant la trace de l'exception du point de vue du fils.

L'exception la plus communément levée est *OSError*. Elle survient, par exemple, si vous essayez d'exécuter un fichier inexistant. Les applications doivent se préparer à traiter des exceptions *OSError*.

Une *ValueError* sera levée si *Popen* est appelé avec des arguments invalides.

check_call() et *check_output()* lèveront une *CalledProcessError* si le processus appelé renvoie un code de retour non nul.

Toutes les fonctions et méthodes qui acceptent un paramètre *timeout*, telles que *call()* et *Popen.communicate()* lèveront une *TimeoutExpired* si le *timeout* expire avant la fin du processus.

Toutes les exceptions définies dans ce module héritent de *SubprocessError*.

Nouveau dans la version 3.3 : Ajout de la classe de base *SubprocessError*.

17.5.2 Considérations de sécurité

Contrairement à quelques autres fonctions *popen*, cette implémentation n'appellera jamais implicitement le *shell* du système. Cela signifie que tous les caractères, incluant les métacaractères des *shells*, peuvent être passés aux processus fils en toute sécurité. Si le *shell* est invoqué explicitement, avec *shell=True*, il est de la responsabilité de l'application d'assurer que les espaces et métacaractères sont échappés correctement pour éviter les vulnérabilités de type *shell injection*.

Avec *shell=True*, la fonction *shlex.quote()* peut être utilisée pour échapper proprement les espaces et métacaractères dans les chaînes qui seront utilisées pour construire les commandes *shell*.

17.5.3 Objets *Popen*

Les instances de la classe *Popen* possèdent les méthodes suivantes :

Popen.poll()

Vérifie que le processus enfant s'est terminé. Modifie et renvoie l'attribut *returncode*, sinon, renvoie *None*.

Popen.wait(timeout=None)

Attend qu'un processus enfant se termine. Modifie l'attribut *returncode* et le renvoie.

Si le processus ne se termine pas après le nombre de secondes spécifié par *timeout*, une exception *TimeoutExpired* est levée. Cela ne pose aucun problème d'attraper cette exception et de réessayer d'attendre.

Note : Cela provoquera un blocage (*deadlock*) lors de l'utilisation de *stdout=PIPE* ou *stderr=PIPE* si le processus fils génère tellement de données sur le tube qu'il le bloque, en attente que le système d'exploitation permette au tampon du tube d'accepter plus de données. Utilisez *Popen.communicate()* pour éviter ce problème lors de l'utilisation de tubes.

Note : Cette fonction est implémentée avec une attente active (appels non bloquants et *sleep* courts). Utilisez le module *asyncio* pour une attente asynchrone : voir *asyncio.create_subprocess_exec*.

Modifié dans la version 3.3 : Ajout de *timeout*.

Obsolète depuis la version 3.4 : N'utilisez pas le paramètre *endtime*. Il a été par mégarde exposé dans la version 3.3 mais laissé non-documenté, et était destiné à rester privé pour un usage interne. Utilisez plutôt *timeout*.

`Popen.communicate(input=None, timeout=None)`

Interagit avec le processus : envoie des données sur l'entrée standard, lit en retour les données sur les sorties standard et d'erreur, et attend que le processus se termine. L'argument optionnel *input* contient les données à envoyer au processus fils, ou `None` s'il n'y a aucune donnée à lui transmettre. Si les flux sont ouverts en mode texte, *input* doit être une chaîne de caractère. Autrement, ce doit être un objet *bytes*.

`communicate()` renvoie un *tuple* (`stdout_data`, `stderr_data`). Les données seront des chaînes de caractères si les flux sont ouverts en mode texte, et des objets *bytes* dans le cas contraire.

Notez que si vous souhaitez envoyer des données sur l'entrée standard du processus, vous devez créer l'objet *Popen* avec `stdin=PIPE`. Similairement, pour obtenir autre chose que `None` dans le *tuple* résultant, vous devez aussi préciser `stdout=PIPE` et/ou `stderr=PIPE`.

Si le processus ne se termine pas après *timeout* secondes, une exception *TimeoutExpired* est levée. Attraper cette exception et retenter la communication ne fait perdre aucune donnée de sortie.

Le processus enfant n'est pas tué si le *timeout* expire, donc afin de nettoyer proprement le tout, une application polie devrait tuer le processus fils et terminer la communication :

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

Note : Les données lues sont mises en cache en mémoire, donc n'utilisez pas cette méthode si la taille des données est importante voire illimitée.

Modifié dans la version 3.3 : Ajout de *timeout*.

`Popen.send_signal(signal)`

Envoie le signal *signal* au fils.

Note : Sous Windows, *SIGTERM* est un alias pour *terminate()*. *CTRL_C_EVENT* et *CTRL_BREAK_EVENT* peuvent être envoyés aux processus démarrés avec un paramètre *creationflags* incluant *CREATE_NEW_PROCESS_GROUP*.

`Popen.terminate()`

Stoppe le processus fils. Sur les systèmes POSIX, la méthode envoie un signal *SIGTERM* au fils. Sous Windows, la fonction `TerminateProcess()` de l'API *Win32* est appelée pour arrêter le fils.

`Popen.kill()`

Tue le processus fils. Sur les systèmes POSIX, la fonction envoie un signal *SIGKILL* au fils. Sous Windows, *kill()* est un alias pour *terminate()*.

Les attributs suivants sont aussi disponibles :

`Popen.args`

L'argument *args* tel que passé à *Popen* – une séquence d'arguments du programme ou une simple chaîne de caractères.

Nouveau dans la version 3.3.

`Popen.stdin`

Si l'argument *stdin* valait *PIPE*, cet attribut est un flux accessible en écriture comme renvoyé par *open()*. Si

les arguments *encoding* ou *errors* ont été spécifiés, ou si *universal_newlines* valait `True`, le flux est textuel, il est autrement binaire. Si l'argument *stdin* ne valait pas *PIPE*, cet attribut est `None`.

`Popen.stdout`

Si l'argument *stdout* valait *PIPE*, cet attribut est un flux accessible en lecture comme renvoyé par `open()`. Lire depuis le flux fournit la sortie du processus fils. Si les arguments *encoding* ou *errors* ont été spécifiés, ou si *universal_newlines* valait `True`, le flux est textuel, il est autrement binaire. Si l'argument *stdout* ne valait pas *PIPE*, cet attribut est `None`.

`Popen.stderr`

Si l'argument *stderr* valait *PIPE*, cet attribut est un flux accessible en lecture comme renvoyé par `open()`. Lire depuis le flux fournit la sortie d'erreur du processus fils. Si les arguments *encoding* ou *errors* ont été spécifiés, ou si *universal_newlines* valait `True`, le flux est textuel, il est autrement binaire. Si l'argument *stderr* ne valait pas *PIPE*, cet attribut est `None`.

Avertissement : Utilisez `communicate()` plutôt que `.stdin.write`, `.stdout.read` ou `.stderr.read` pour empêcher les *deadlocks* dus au remplissage des tampons des tubes de l'OS et bloquant le processus enfant.

`Popen.pid`

L'identifiant de processus du processus enfant.

Notez que si vous passez l'argument *shell* à `True`, il s'agit alors de l'identifiant du *shell* instancié.

`Popen.returncode`

Le code de retour de l'enfant, attribué par `poll()` et `wait()` (et indirectement par `communicate()`). Une valeur `None` indique que le processus ne s'est pas encore terminé.

Une valeur négative `-N` indique que le processus enfant a été terminé par un signal `N` (seulement sur les systèmes *POSIX*).

17.5.4 Utilitaires *Popen* pour Windows

La classe `STARTUPINFO` et les constantes suivantes sont seulement disponibles sous Windows.

`class subprocess.STARTUPINFO`

Un support partiel de la structure `STARTUPINFO` de Windows est utilisé lors de la création d'un objet *Popen*.

`dwFlags`

Un champ de bits déterminant si certains attributs `STARTUPINFO` sont utilisés quand le processus crée une fenêtre

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

`hStdInput`

Si `dwFlags` spécifie `STARTF_USESTDHANDLES`, cet attribut est le descripteur d'entrée standard du processus. Si `STARTF_USESTDHANDLES` n'est pas spécifié, l'entrée standard par défaut est le tampon du clavier.

`hStdOutput`

Si `dwFlags` spécifie `STARTF_USESTDHANDLES`, cet attribut est le descripteur de sortie standard du processus. Autrement, l'attribut est ignoré et la sortie standard par défaut est le tampon de la console.

`hStdError`

Si `dwFlags` spécifie `STARTF_USESTDHANDLES`, cet attribut est le descripteur de sortie d'erreur du processus. Autrement, l'attribut est ignoré et la sortie d'erreur par défaut est le tampon de la console.

`wShowWindow`

Si `dwFlags` spécifie `STARTF_USESHOWWINDOW`, cet attribut peut-être n'importe quel attribut valide pour

le paramètre `nCmdShow` de la fonction `ShowWindow`, à l'exception de `SW_SHOWDEFAULT`. Autrement, cet attribut est ignoré.

`SW_HIDE` est fourni pour cet attribut. Il est utilisé quand `Popen` est appelée avec `shell=True`.

Constantes

Le module `subprocess` expose les constantes suivantes.

`subprocess.STD_INPUT_HANDLE`

Le périphérique d'entrée standard. Initialement, il s'agit du tampon de la console d'entrée, `CONIN$`.

`subprocess.STD_OUTPUT_HANDLE`

Le périphérique de sortie standard. Initialement, il s'agit du tampon de l'écran de console actif, `CONOUT$`.

`subprocess.STD_ERROR_HANDLE`

Le périphérique de sortie d'erreur. Initialement, il s'agit du tampon de l'écran de console actif, `CONOUT$`.

`subprocess.SW_HIDE`

Cache la fenêtre. Une autre fenêtre sera activée.

`subprocess.STARTF_USESTDHANDLES`

Spécifie que les attributs `STARTUPINFO.hStdInput`, `STARTUPINFO.hStdOutput` et `STARTUPINFO.hStdError` contiennent des informations additionnelles.

`subprocess.STARTF_USESHOWWINDOW`

Spécifie que l'attribut `STARTUPINFO.wShowWindow` contient des informations additionnelles.

`subprocess.CREATE_NEW_CONSOLE`

Le nouveau processus instancie une nouvelle console, plutôt que d'hériter de celle de son père (par défaut).

`subprocess.CREATE_NEW_PROCESS_GROUP`

Paramètre `creationflags` de `Popen` pour spécifier si un nouveau groupe de processus doit être créé. Cette option est nécessaire pour utiliser `os.kill()` sur le sous-processus.

L'option est ignorée si `CREATE_NEW_CONSOLE` est spécifié.

17.5.5 Ancienne interface (API) haut-niveau

Avant Python 3.5, ces trois fonctions représentaient l'API haut-niveau de `subprocess`. Vous pouvez maintenant utiliser `run()` dans de nombreux cas, mais beaucoup de codes existant font appel à ces trois fonctions.

`subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None)`

Lance la commande décrite par `args`, attend qu'elle se termine, et renvoie son attribut `returncode`.

C'est équivalent à :

```
run(...).returncode
```

(excepté que les paramètres `input` et `check` ne sont pas supportés)

Les arguments montrés plus haut sont sûrement les plus communs. La signature complète de la fonction est en grande partie la même que le constructeur de `Popen` : cette fonction passe tous les arguments fournis autre que `timeout` directement à travers cette interface.

Note : N'utilisez pas `stdout=PIPE` ou `stderr=PIPE` avec cette fonction. Le processus enfant bloquera s'il génère assez de données pour remplir le tampon du tube de l'OS, puisque les tubes ne seront jamais lus.

Modifié dans la version 3.3 : Ajout de `timeout`.

`subprocess.check_call` (*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*)

Lance la commande avec les arguments et attend qu'elle se termine. Se termine normalement si le code de retour est zéro, et lève une `CalledProcessError` autrement. L'objet `CalledProcessError` contiendra le code de retour dans son attribut `returncode`.

C'est équivalent à :

```
run(..., check=True)
```

(excepté que le paramètre `input` n'est pas pris en charge)

Les arguments montrés plus haut sont sûrement les plus communs. La signature complète de la fonction est en grande partie la même que le constructeur de `Popen` : cette fonction passe tous les arguments fournis autre que `timeout` directement à travers cette interface.

Note : N'utilisez pas `stdout=PIPE` ou `stderr=PIPE` avec cette fonction. Le processus enfant bloquera s'il génère assez de données pour remplir le tampon du tube de l'OS, puisque les tubes ne seront jamais lus.

Modifié dans la version 3.3 : Ajout de `timeout`.

`subprocess.check_output` (*args*, *, *stdin=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *errors=None*, *universal_newlines=False*, *timeout=None*)

Lance la commande avec les arguments et renvoie sa sortie.

Si le code de retour est non-nul, la fonction lève une `CalledProcessError`. L'objet `CalledProcessError` contiendra le code de retour dans son attribut `returncode`, et la sortie du programme dans son attribut `output`.

C'est équivalent à :

```
run(..., check=True, stdout=PIPE).stdout
```

Les arguments montrés plus haut sont sûrement les plus communs. La signature complète de la fonction est en grande partie la même que `run()` : la plupart des arguments sont passés directement par cette interface. Cependant, passer explicitement `input=None` pour hériter du gestionnaire d'entrée standard du parent n'est pas supporté.

Par défaut, cette fonction renvoie les données encodées sous forme de `bytes`. Le réel encodage des données de sortie peut dépendre de la commande invoquée, donc le décodage du texte devra souvent être géré au niveau de l'application.

Ce comportement peut être redéfini en mettant `universal_newlines` à `True` comme décrit ci-dessus dans [Arguments fréquemment utilisés](#).

Pour capturer aussi la sortie d'erreur dans le résultat, utilisez `stderr=subprocess.STDOUT` :

```
>>> subprocess.check_output(  
...     "ls non_existent_file; exit 0",  
...     stderr=subprocess.STDOUT,  
...     shell=True)  
'ls: non_existent_file: No such file or directory\n'
```

Nouveau dans la version 3.1.

Modifié dans la version 3.3 : Ajout de `timeout`.

Modifié dans la version 3.4 : Ajout de la gestion de l'argument nommé `input`.

Modifié dans la version 3.6 : Ajout d'`encoding` et `errors`. Consultez `run()` pour plus d'informations.

17.5.6 Remplacer les fonctions plus anciennes par le module `subprocess`

Dans cette section, « a devient b » signifie que b peut être utilisée en remplacement de a.

Note : Toutes les fonctions « a » dans cette section échouent (plus ou moins) silencieusement si le programme à exécuter ne peut être trouvé ; les fonctions « b » de remplacement lèvent à la place une `OSError`.

De plus, les remplacements utilisant `check_output()` échoueront avec une `CalledProcessError` si l'opération requise produit un code de retour non-nul. La sortie est toujours disponible par l'attribut `output` de l'exception levée.

Dans les exemples suivants, nous supposons que les fonctions utilisées ont déjà été importées depuis le module `subprocess`.

Remplacement des *backquotes* des *shells* `/bin/sh`

```
output=`mycmd myarg`
```

devient :

```
output = check_output(["mycmd", "myarg"])
```

Remplacer les *pipes* du *shell*

```
output=`dmesg | grep hda`
```

devient :

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close() # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

L'appel à `p1.stdout.close()` après le démarrage de `p2` est important pour que `p1` reçoive un `SIGPIPE` si `p2` se termine avant lui.

Alternativement, pour des entrées fiables, la gestion des tubes du *shell* peut directement être utilisé :

```
output=`dmesg | grep hda`
```

devient :

```
output=check_output("dmesg | grep hda", shell=True)
```

Remplacer `os.system()`

```
sts = os.system("mycmd" + " myarg")
# becomes
sts = call("mycmd" + " myarg", shell=True)
```

Notes :

— Appeler le programme à travers un *shell* n'est habituellement pas requis.

Un exemple plus réaliste ressemblerait à cela :

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

Remplacer les fonctions de la famille `os.spawn`

Exemple avec `P_NOWAIT` :

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

Exemple avec `P_WAIT` :

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Exemple avec un tableau :

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Exemple en passant un environnement :

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```


Remplacer `os.popen()`, `os.popen2()`, `os.popen3()`

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

La gestion du code de retour se traduit comme suit :

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

Remplacer les fonctions du module `popen2`

Note : Si l'argument *cmd* des fonctions de *popen2* est une chaîne de caractères, la commande est exécutée à travers */bin/sh*. Si c'est une liste, la commande est directement exécutée.

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
```

(suite sur la page suivante)

(suite de la page précédente)

```

        stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)

```

`popen2.Popen3` et `popen2.Popen4` fonctionnent basiquement comme `subprocess.Popen`, excepté que :

- `Popen` lève une exception si l'exécution échoue.
- L'argument `capturestderr` est remplacé par `stderr`.
- `stdin=PIPE` et `stdout=PIPE` doivent être spécifiés.
- `popen2` ferme tous les descripteurs de fichiers par défaut, mais vous devez spécifier `close_fds=True` avec `Popen` pour garantir ce comportement sur toutes les plateformes ou les anciennes versions de Python.

17.5.7 Remplacement des fonctions originales d'invocation du *shell*

Ce module fournit aussi les fonctions suivantes héritées du module `commands` de Python 2.x. Ces opérations invoquent implicitement le *shell* du système et n'apportent aucune des garanties décrites ci-dessus par rapport à la sécurité ou la cohérence de la gestion des exceptions.

`subprocess.getstatusoutput(cmd)`

Renvoie les valeurs (`exitcode`, `output`) de l'exécution de `cmd` dans un *shell*.

Exécute la chaîne `cmd` dans un *shell* avec `Popen.check_output()` et renvoie un *tuple* de 2 éléments (`exitcode`, `output`). L'encodage local est utilisé, voir les notes de la section [Arguments fréquemment utilisés](#) pour plus de détails.

Si la sortie se termine par un caractère de fin de ligne, ce dernier est supprimé. Le code de statut de la commande peut être interprété comme le code de retour de `subprocess`. Par exemple :

```

>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')

```

Disponibilité : POSIX et Windows

Modifié dans la version 3.3.4 : Ajout de la gestion de Windows.

The function now returns (`exitcode`, `output`) instead of (`status`, `output`) as it did in Python 3.3.3 and earlier. `exitcode` has the same value as [returncode](#).

`subprocess.getoutput(cmd)`

Renvoie la sortie (standard et d'erreur) de l'exécution de `cmd` dans un *shell*.

Like [getstatusoutput\(\)](#), except the exit code is ignored and the return value is a string containing the command's output. Example :

```

>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'

```

Disponibilité : POSIX et Windows

Modifié dans la version 3.3.4 : Ajout de la gestion de Windows

17.5.8 Notes

Convertir une séquence d'arguments vers une chaîne de caractères sous Windows

Sous Windows, une séquence *args* est convertie vers une chaîne qui peut être analysée avec les règles suivantes (qui correspondent aux règles utilisées par l'environnement *MS C*) :

1. Les arguments sont délimités par des espacements, qui peuvent être des espaces ou des tabulations.
2. Une chaîne entourée de double guillemets est interprétée comme un argument seul, qu'elle contienne ou non des espacements. Une chaîne entre guillemets peut être intégrée dans un argument.
3. Un guillemet double précédé d'un *backslash* est interprété comme un guillemet double littéral.
4. Les *backslashes* sont interprétés littéralement, à moins qu'ils précèdent immédiatement un guillemet double.
5. Si des *backslashes* précèdent directement un guillemet double, toute paire de *backslashes* est interprétée comme un *backslash* littéral. Si le nombre de *backslashes* est impair, le dernier *backslash* échappe le prochain guillemet double comme décrit en règle 3.

Voir aussi :

shlex Module qui fournit des fonctions pour analyser et échapper les lignes de commandes.

17.6 sched — Event scheduler

Code source : [Lib/sched.py](https://lib.sched.py)

The *sched* module defines a class which implements a general purpose event scheduler :

class `sched.scheduler` (*timefunc=time.monotonic, delayfunc=time.sleep*)

The *scheduler* class defines a generic interface to scheduling events. It needs two functions to actually deal with the « outside world » — *timefunc* should be callable without arguments, and return a number (the « time », in any units whatsoever). If *time.monotonic* is not available, the *timefunc* default is *time.time* instead. The *delayfunc* function should be callable with one argument, compatible with the output of *timefunc*, and should delay that many time units. *delayfunc* will also be called with the argument 0 after each event is run to allow other threads an opportunity to run in multi-threaded applications.

Modifié dans la version 3.3 : *timefunc* and *delayfunc* parameters are optional.

Modifié dans la version 3.3 : *scheduler* class can be safely used in multi-threaded environments.

Exemple :

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.run()
...     print(time.time())
...
>>> print_some_times()
```

(suite sur la page suivante)

```

930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276

```

17.6.1 Scheduler Objects

`scheduler` instances have the following methods and attributes :

`scheduler.enterabs` (*time*, *priority*, *action*, *argument*=(), *kwargs*={})

Schedule a new event. The *time* argument should be a numeric type compatible with the return value of the *timefunc* function passed to the constructor. Events scheduled for the same *time* will be executed in the order of their *priority*. A lower number represents a higher priority.

Executing the event means executing *action* (**argument*, ***kwargs*). *argument* is a sequence holding the positional arguments for *action*. *kwargs* is a dictionary holding the keyword arguments for *action*.

Return value is an event which may be used for later cancellation of the event (see `cancel()`).

Modifié dans la version 3.3 : *argument* parameter is optional.

Nouveau dans la version 3.3 : *kwargs* parameter was added.

`scheduler.enter` (*delay*, *priority*, *action*, *argument*=(), *kwargs*={})

Schedule an event for *delay* more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for `enterabs()`.

Modifié dans la version 3.3 : *argument* parameter is optional.

Nouveau dans la version 3.3 : *kwargs* parameter was added.

`scheduler.cancel` (*event*)

Remove the event from the queue. If *event* is not an event currently in the queue, this method will raise a `ValueError`.

`scheduler.empty` ()

Return true if the event queue is empty.

`scheduler.run` (*blocking*=True)

Run all scheduled events. This method will wait (using the `delayfunc()` function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

If *blocking* is false executes the scheduled events due to expire soonest (if any) and then return the deadline of the next scheduled call in the scheduler (if any).

Either *action* or *delayfunc* can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by *action*, the event will not be attempted in future calls to `run()`.

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent.

Nouveau dans la version 3.3 : *blocking* parameter was added.

`scheduler.queue`

Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a *named tuple* with the following fields : *time*, *priority*, *action*, *argument*, *kwargs*.

17.7 queue — File synchronisée

Code source : [Lib/queue.py](#)

Le module `queue` implémente des files multi-productrices et multi-consommatrices. C'est particulièrement utile en programmation *multi-thread*, lorsque les informations doivent être échangées sans risques entre plusieurs *threads*. La classe `Queue` de ce module implémente tout le verrouillage nécessaire. Cela dépend de la disponibilité du support des fils d'exécution en Python ; voir le module `threading`.

Le module implémente trois types de files qui diffèrent par l'ordre dans lequel les éléments en sont extraits. Dans une file FIFO, les premiers éléments ajoutés sont les premiers extraits. Dans une file LIFO, le dernier élément ajouté est le premier élément extrait (se comporte comme une pile). Avec une file de priorité, les entrées restent triés (en utilisant le module `heapq`) et l'élément ayant la valeur la plus faible est extrait en premier.

En interne, le module utilise des verrous pour bloquer temporairement des fils d'exécution concurrents. Cependant, il n'a pas été conçu pour être réentrant au sein d'un fil d'exécution.

Le module `queue` définit les classes et les exceptions suivantes :

class `queue.Queue` (*maxsize=0*)

Constructeur pour une file FIFO. *maxsize* est un entier définissant le nombre maximal d'éléments pouvant être mis dans la file. L'insertion sera bloquée lorsque cette borne supérieure sera atteinte, jusqu'à ce que des éléments de la file soient consommés. Si *maxsize* est inférieur ou égal à 0, la taille de la file sera infinie.

class `queue.LifoQueue` (*maxsize=0*)

Constructeur pour une file LIFO. *maxsize* est un entier définissant le nombre maximal d'éléments pouvant être mis dans la file. L'insertion sera bloquée lorsque cette borne supérieure sera atteinte, jusqu'à ce que des éléments de la file soient consommés. Si *maxsize* est inférieur ou égal à 0, la taille de la file sera infinie.

class `queue.PriorityQueue` (*maxsize=0*)

Constructeur pour une file de priorité. *maxsize* est un entier définissant le nombre maximal d'éléments pouvant être mis dans la file. L'insertion sera bloquée lorsque cette borne supérieure sera atteinte, jusqu'à ce que des éléments soient consommés. Si *maxsize* est inférieur ou égal à 0, la taille de la file sera infinie.

Les éléments de valeurs les plus faibles sont extraits en premier (l'élément de valeur la plus faible est celui renvoyé par `sorted(list(entries))[0]`). Un cas typique est d'utiliser des tuple de la forme : (*priority_number*, *data*).

exception `queue.Empty`

Exception levée lorsque la méthode non bloquante `get()` (ou `get_nowait()`) est appelée sur l'objet `Queue` vide.

exception `queue.Full`

Exception levée lorsque la méthode non bloquante `put()` (ou `put_nowait()`) est appelée sur un objet `Queue` plein.

17.7.1 Objets Queue

Les objets `Queue` (`Queue`, `LifoQueue` ou `PriorityQueue`) fournissent les méthodes publiques décrites ci-dessous.

`Queue.qsize()`

Renvoie la taille approximative de la file. Notez que `qsize() > 0` ne garantit pas qu'un `get()` ultérieur ne sera pas bloquant et que `qsize() < maxsize` ne garantit pas non plus qu'un `put()` ne sera pas bloquant.

`Queue.empty()`

Renvoie `True` si la file est vide, `False` sinon. Si `empty()` renvoie `True`, cela ne garantit pas qu'un appel ultérieur à `put()` ne sera pas bloquant. Similairement, si `empty()` renvoie `False`, cela ne garantit pas qu'un appel ultérieur à `get()` ne sera pas bloquant.

`Queue.full()`

Renvoie `True` si la file est pleine, `False` sinon. Si `full()` renvoie `True`, cela ne garantit pas qu'un appel ultérieur à `get()` ne sera pas bloquant. Similairement, si `full()` retourne `False`, cela ne garantit pas qu'un appel ultérieur à `put()` ne sera pas bloquant.

`Queue.put(item, block=True, timeout=None)`

Met `item` dans la file. Si les arguments optionnels `block` et `timeout` sont respectivement `True` et `None` (les valeurs par défaut), la méthode bloque si nécessaire jusqu'à ce qu'un emplacement libre soit disponible. Si `timeout` est un nombre positif, elle bloque au plus `timeout` secondes et lève l'exception `Full` s'il n'y avait pas d'emplacement libre pendant cette période de temps. Sinon (`block` est `False`), elle met un élément dans la file s'il y a un emplacement libre immédiatement disponible. Si ce n'est pas le cas, elle lève l'exception `Full` (`timeout` est ignoré dans ce cas).

`Queue.put_nowait(item)`

Équivalent à `put(item, False)`.

`Queue.get(block=True, timeout=None)`

Retire et renvoie un élément de la file. Si les arguments optionnels `block` et `timeout` valent respectivement `True` et `None` (les valeurs par défaut), la méthode bloque si nécessaire jusqu'à ce qu'un élément soit disponible. Si `timeout` est un entier positif, elle bloque au plus `timeout` secondes et lève l'exception `Empty` s'il n'y avait pas d'élément disponible pendant cette période de temps. Sinon (`block` vaut `False`), elle renvoie un élément s'il y en a un immédiatement disponible. Si ce n'est pas le cas, elle lève l'exception `Empty` (`timeout` est ignoré dans ce cas).

`Queue.get_nowait()`

Équivalent à `get(False)`.

Deux méthodes sont proposées afin de savoir si les tâches mises dans la file ont été entièrement traitées par les fils d'exécution consommateurs du démon.

`Queue.task_done()`

Indique qu'une tâche précédemment mise dans la file est terminée. Utilisé par les fils d'exécution consommateurs de la file. Pour chaque appel à `get()` effectué afin de récupérer une tâche, un appel ultérieur à `task_done()` informe la file que le traitement de la tâche est terminé.

Si un `join()` est actuellement bloquant, on reprendra lorsque tous les éléments auront été traités (ce qui signifie qu'un appel à `task_done()` a été effectué pour chaque élément qui a été `put()` dans la file).

Lève une exception `ValueError` si appelée plus de fois qu'il y avait d'éléments dans la file.

`Queue.join()`

Bloque jusqu'à ce que tous les éléments de la file aient été obtenus et traités.

Le nombre de tâches inachevées augmente chaque fois qu'un élément est ajouté à la file. Ce nombre diminue chaque fois qu'un fil d'exécution consommateur appelle `task_done()` pour indiquer que l'élément a été extrait et que tout le travail à effectuer dessus est terminé. Lorsque le nombre de tâches non terminées devient nul, `join()` débloque.

Exemple montrant comment attendre que les tâches mises dans la file soient terminées :

```
def worker():
    while True:
        item = q.get()
        if item is None:
            break
        do_work(item)
        q.task_done()
```

(suite sur la page suivante)

(suite de la page précédente)

```

q = queue.Queue()
threads = []
for i in range(num_worker_threads):
    t = threading.Thread(target=worker)
    t.start()
    threads.append(t)

for item in source():
    q.put(item)

# block until all tasks are done
q.join()

# stop workers
for i in range(num_worker_threads):
    q.put(None)
for t in threads:
    t.join()

```

Voir aussi :

Classe `multiprocessing.Queue` Une file à utiliser dans un contexte multi-processus (plutôt que *multi-thread*).

`collections.deque` est une implémentation alternative de file non bornée avec des méthodes `append()` et `popleft()` rapides et atomiques ne nécessitant pas de verrouillage.

Les modules suivants servent de fondation pour certains services cités ci-dessus.

17.8 dummy_threading — Module de substitution au module threading

Code source : [Lib/dummy_threading.py](#)

Ce module fournit un duplicat de l'interface du module `threading`. Son but est d'être importé lorsque le module `_thread` n'est pas fourni par la plateforme.

Utilisation suggérée :

```

try:
    import threading
except ImportError:
    import dummy_threading as threading

```

Soyez prudent de ne pas utiliser ce module lorsqu'un interblocage (*deadlock* en anglais) peut se produire à partir d'un fil d'exécution en cours de création qui bloque en attendant qu'un autre fil d'exécution soit créé. Cela se produit souvent avec des I/O bloquants.

17.9 `_thread` — API bas niveau de gestion de fils d'exécution

Ce module fournit les primitives de bas niveau pour travailler avec de multiples fils d'exécution (aussi appelés *light-weight processes* ou *tasks*) — plusieurs fils d'exécution de contrôle partagent leur espace de données global. Pour la synchronisation, de simples verrous (aussi appelés des *mutexes* ou des *binary semaphores*) sont fournis. Le module `threading` fournit une API de fils d'exécution de haut niveau, plus facile à utiliser et construite à partir de ce module.

The module is optional. It is supported on Windows, Linux, SGI IRIX, Solaris 2.x, as well as on systems that have a POSIX thread (a.k.a. « pthread ») implementation. For systems lacking the `_thread` module, the `_dummy_thread` module is available. It duplicates this module's interface and can be used as a drop-in replacement.

Elle définit les constantes et fonctions suivantes :

exception `_thread.error`

Levée lors d'erreurs spécifiques aux fils d'exécution.

Modifié dans la version 3.3 : Ceci est à présent un synonyme de l'exception native `RuntimeError`.

`_thread.LockType`

C'est le type d'objets verrous.

`_thread.start_new_thread (function, args[, kwargs])`

Démarre un nouveau fils d'exécution et renvoie son identifiant. Ce fil d'exécution exécute la fonction *function* avec la liste d'arguments *args* (qui doit être un *tuple*). L'argument optionnel *kwargs* spécifie un dictionnaire d'arguments de mots clés. Quand la fonction se termine, le fil d'exécution se termine silencieusement. Quand la fonction termine avec une exception non gérée, une trace de la pile est affichée et ensuite le fil d'exécution s'arrête (mais les autres fils d'exécutions continuent de s'exécuter).

`_thread.interrupt_main ()`

Raises a `KeyboardInterrupt` exception in the main thread. A subthread can use this function to interrupt the main thread.

`_thread.exit ()`

Lève une exception `SystemExit`. Quand elle n'est pas interceptée, le fil d'exécution se terminera silencieusement.

`_thread.allocate_lock ()`

Renvoie un nouveau verrou. Les méthodes des verrous sont décrites ci-dessous. Le verrou est initialement déverrouillé.

`_thread.get_ident ()`

Renvoie l'« identifiant de fil » du fil d'exécution courant. C'est un entier non nul. Sa valeur n'a pas de signification directe ; il est destiné à être utilisé comme *cookie* magique, par exemple pour indexer un dictionnaire de données pour chaque fil. Les identifiants de fils peuvent être recyclés lorsqu'un fil d'exécution se termine et qu'un autre fil est créé.

`_thread.stack_size ([size])`

Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a `RuntimeError` is raised. If the specified stack size is invalid, a `ValueError` is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common ; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability : Windows, systems with POSIX threads.

`_thread.TIMEOUT_MAX`

La valeur maximale autorisée pour le paramètre *timeout* de la méthode `Lock.acquire()`. Préciser un délai d'attente supérieur à cette valeur lève une exception `OverflowError`.

Nouveau dans la version 3.2.

Les verrous ont les méthodes suivantes :

`lock.acquire(waitflag=1, timeout=-1)`

Sans aucun argument optionnel, cette méthode acquiert le verrou inconditionnellement, et si nécessaire attend jusqu'à ce qu'il soit relâché par un autre fil d'exécution (un seul fil d'exécution à la fois peut acquérir le verrou — c'est leur raison d'être).

Si l'argument *waitflag*, un entier, est présent, l'action dépend de sa valeur : si elle est de zéro, le verrou est seulement acquis s'il peut être acquis immédiatement, sans attendre, sinon le verrou est acquis inconditionnellement comme ci-dessus.

Si l'argument *timeout*, en virgule flottante, est présent et positif, il spécifie le temps d'attente maximum en secondes avant de renvoyer. Un argument *timeout* négatif spécifie une attente illimitée. Vous ne pouvez pas spécifier un *timeout* si *waitflag* est à zéro.

La valeur renvoyée est `True` si le verrou est acquis avec succès, sinon `False`.

Modifié dans la version 3.2 : Le paramètre *timeout* est nouveau.

Modifié dans la version 3.2 : Le verrou acquis peut maintenant être interrompu par des signaux sur POSIX.

`lock.release()`

Relâche le verrou. Le verrou doit avoir été acquis plus tôt, mais pas nécessairement par le même fil d'exécution.

`lock.locked()`

Renvoie le statut du verrou : `True` s'il a été acquis par certains fils d'exécution, sinon `False`.

En plus de ces méthodes, les objets verrous peuvent aussi être utilisés via l'instruction `with`, e.g. :

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

Avertissements :

- Les fils d'exécution interagissent étrangement avec les interruptions : l'exception `KeyboardInterrupt` sera reçue par un fil d'exécution arbitraire. (Quand le module `signal` est disponible, les interruptions vont toujours au fil d'exécution principal).
- Appeler la fonction `sys.exit()` ou lever l'exception `SystemExit` est équivalent à appeler la fonction `_thread.exit()`.
- Il n'est pas possible d'interrompre la méthode `acquire()` sur un verrou — l'exception `KeyboardInterrupt` surviendra après que le verrou a été acquis.
- Quand le fil d'exécution principal s'arrête, il est défini par le système si les autres fils d'exécution survivent. Sur beaucoup de systèmes, ils sont tués sans l'exécution des clauses `try... finally` ou l'exécution des destructeurs d'objets.
- Quand le fil d'exécution principal s'arrête, il ne fait pas son nettoyage habituel (excepté que les clauses `try... finally` sont honorées) et les fichiers d'entrée/sortie standards ne sont pas nettoyés.

17.10 `_dummy_thread` — Module de substitution pour le module `_thread`

Code source : [Lib/_dummy_thread.py](#)

Ce module fournit un répliat de l'interface du module `_thread`. Son but est d'être importé lorsque le module `_thread` n'est pas fournit par la plateforme.

Utilisation suggérée :

```
try:
    import _thread
except ImportError:
    import _dummy_thread as _thread
```

Soyez prudent de ne pas utiliser ce module lorsqu'un interblocage (*deadlock* en anglais) peut se produire à partir d'un fil d'exécution en cours de création qui bloque en attendant qu'un autre fil d'exécution soit créé. Cela se produit souvent avec des I/O bloquants.

Communication et réseau entre processus

Les modules décrits dans ce chapitre fournissent différents mécanismes permettant à des processus de communiquer.

Certains de ces modules ne fonctionnent que pour deux processus sur une seule machine, comme les modules *signal* et *mmap*. D'autres gèrent des protocoles réseaux que deux processus, ou plus, peuvent utiliser pour communiquer entre différentes machines.

La liste des modules documentés dans ce chapitre est :

18.1 *socket* — Low-level networking interface

Source code : [Lib/socket.py](#)

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Note : Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style : the *socket()* function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface : as with *read()* and *write()* operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

Voir aussi :

Module *socketserver* Classes that simplify writing network servers.

Module *ssl* A TLS/SSL wrapper for socket objects.

18.1.1 Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows :

- The address of an `AF_UNIX` socket bound to a file system node is represented as a string, using the file system encoding and the `'surrogateescape'` error handler (see [PEP 383](#)). An address in Linux's abstract namespace is returned as a *bytes-like object* with an initial null byte ; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

Modifié dans la version 3.3 : Previously, `AF_UNIX` socket paths were assumed to use UTF-8 encoding.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

- A pair (`host`, `port`) is used for the `AF_INET` address family, where `host` is a string representing either a hostname in Internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and `port` is an integer.
 - For IPv4 addresses, two special forms are accepted instead of a host address : `' '` represents `INADDR_ANY`, which is used to bind to all interfaces, and the string `'<broadcast>'` represents `INADDR_BROADCAST`. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.
- For `AF_INET6` address family, a four-tuple (`host`, `port`, `flowinfo`, `scopeid`) is used, where `flowinfo` and `scopeid` represent the `sin6_flowinfo` and `sin6_scope_id` members in struct `sockaddr_in6` in C. For `socket` module methods, `flowinfo` and `scopeid` can be omitted just for backward compatibility. Note, however, omission of `scopeid` can cause problems in manipulating scoped IPv6 addresses.
- `AF_NETLINK` sockets are represented as pairs (`pid`, `groups`).
- Linux-only support for TIPC is available using the `AF_TIPC` address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is (`addr_type`, `v1`, `v2`, `v3` [, `scope`]), where :
 - `addr_type` is one of `TIPC_ADDR_NAMESEQ`, `TIPC_ADDR_NAME`, or `TIPC_ADDR_ID`.
 - `scope` is one of `TIPC_ZONE_SCOPE`, `TIPC_CLUSTER_SCOPE`, and `TIPC_NODE_SCOPE`.
 - If `addr_type` is `TIPC_ADDR_NAME`, then `v1` is the server type, `v2` is the port identifier, and `v3` should be 0. If `addr_type` is `TIPC_ADDR_NAMESEQ`, then `v1` is the server type, `v2` is the lower port number, and `v3` is the upper port number.
 - If `addr_type` is `TIPC_ADDR_ID`, then `v1` is the node, `v2` is the reference, and `v3` should be set to 0.
- A tuple (`interface`,) is used for the `AF_CAN` address family, where `interface` is a string representing a network interface name like `'can0'`. The network interface name `' '` can be used to receive packets from all network interfaces of this family.
- A string or a tuple (`id`, `unit`) is used for the `SYSPROTO_CONTROL` protocol of the `PF_SYSTEM` family. The string is the name of a kernel control using a dynamically-assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

Nouveau dans la version 3.3.
- `AF_BLUETOOTH` supports the following protocols and address formats :
 - `BTPROTO_L2CAP` accepts (`bdaddr`, `psm`) where `bdaddr` is the Bluetooth address as a string and `psm` is an integer.
 - `BTPROTO_RFCOMM` accepts (`bdaddr`, `channel`) where `bdaddr` is the Bluetooth address as a string and `channel` is an integer.
 - `BTPROTO_HCI` accepts (`device_id`,) where `device_id` is either an integer or a string with the Bluetooth address of the interface. (This depends on your OS ; NetBSD and DragonFlyBSD expect a Bluetooth address while everything else expects an integer.)

Modifié dans la version 3.2 : NetBSD and DragonFlyBSD support added.
 - `BTPROTO_SCO` accepts `bdaddr` where `bdaddr` is a *bytes* object containing the Bluetooth address in a string format. (ex. `b'12:23:34:45:56:67'`) This protocol is not supported under FreeBSD.

- `AF_ALG` is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements (`type`, `name` [, `feat` [, `mask`]]), where :
 - `type` is the algorithm type as string, e.g. `aead`, `hash`, `skcipher` or `rng`.
 - `name` is the algorithm name and operation mode as string, e.g. `sha256`, `hmac (sha256)`, `cbc (aes)` or `drbg_nopr_ctr_aes256`.
 - `feat` and `mask` are unsigned 32bit integers.
 Availability Linux 2.6.38, some algorithm types require more recent Kernels.
 Nouveau dans la version 3.6.
- `AF_PACKET` is a low-level interface directly to network devices. The packets are represented by the tuple (`ifname`, `proto` [, `pkttype` [, `hatype` [, `addr`]]) where :
 - `ifname` - String specifying the device name.
 - `proto` - An in network-byte-order integer specifying the Ethernet protocol number.
 - `pkttype` - Optional integer specifying the packet type :
 - `PACKET_HOST` (the default) - Packet addressed to the local host.
 - `PACKET_BROADCAST` - Physical-layer broadcast packet.
 - `PACKET_MULTICAST` - Packet sent to a physical-layer multicast address.
 - `PACKET_OTHERHOST` - Packet to some other host that has been caught by a device driver in promiscuous mode.
 - `PACKET_OUTGOING` - Packet originating from the local host that is looped back to a packet socket.
 - `hatype` - Optional integer specifying the ARP hardware address type.
 - `addr` - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.

If you use a hostname in the `host` portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in `host` portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised ; starting from Python 3.3, errors related to socket or address semantics raise `OSError` or one of its subclasses (they used to raise `socket.error`).

Non-blocking mode is supported through `setblocking()`. A generalization of this based on timeouts is supported through `settimeout()`.

18.1.2 Module contents

The module `socket` exports the following elements.

Exceptions

exception `socket.error`

A deprecated alias of `OSError`.

Modifié dans la version 3.3 : Following **PEP 3151**, this class was made an alias of `OSError`.

exception `socket.herror`

A subclass of `OSError`, this exception is raised for address-related errors, i.e. for functions that use `h_errno` in the POSIX C API, including `gethostbyname_ex()` and `gethostbyaddr()`. The accompanying value is a pair (`h_errno`, `string`) representing an error returned by a library call. `h_errno` is a numeric value, while `string` represents the description of `h_errno`, as returned by the `hstrerror()` C function.

Modifié dans la version 3.3 : This class was made a subclass of `OSError`.

exception `socket.gaierror`

A subclass of `OSError`, this exception is raised for address-related errors by `getaddrinfo()` and

`getnameinfo()`. The accompanying value is a pair (`error`, `string`) representing an error returned by a library call. `string` represents the description of `error`, as returned by the `gai_strerror()` C function. The numeric `error` value will match one of the `EAI_*` constants defined in this module.

Modifié dans la version 3.3 : This class was made a subclass of `OSError`.

exception `socket.timeout`

A subclass of `OSError`, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to `settimeout()` (or implicitly through `setdefaulttimeout()`). The accompanying value is a string whose value is currently always « timed out ».

Modifié dans la version 3.3 : This class was made a subclass of `OSError`.

Constantes

The `AF_*` and `SOCK_*` constants are now `AddressFamily` and `SocketKind` `IntEnum` collections.

Nouveau dans la version 3.4.

`socket.AF_UNIX`

`socket.AF_INET`

`socket.AF_INET6`

These constants represent the address (and protocol) families, used for the first argument to `socket()`. If the `AF_UNIX` constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

`socket.SOCK_STREAM`

`socket.SOCK_DGRAM`

`socket.SOCK_RAW`

`socket.SOCK_RDM`

`socket.SOCK_SEQPACKET`

These constants represent the socket types, used for the second argument to `socket()`. More constants may be available depending on the system. (Only `SOCK_STREAM` and `SOCK_DGRAM` appear to be generally useful.)

`socket.SOCK_CLOEXEC`

`socket.SOCK_NONBLOCK`

These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

Voir aussi :

[Secure File Descriptor Handling](#) for a more thorough explanation.

Availability : Linux >= 2.6.27.

Nouveau dans la version 3.2.

`SO_*`

`socket.SOMAXCONN`

`MSG_*`

`SOL_*`

`SCM_*`

`IPPROTO_*`

`IPPORT_*`

`INADDR_*`

`IP_*`

`IPV6_*`

`EAI_*`

`AI_*`

`NI_*`

TCP_*

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the `socket` module. They are generally used in arguments to the `setsockopt()` and `getsockopt()` methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

Modifié dans la version 3.6 : `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, `SO_PASSSEC`, `TCP_USER_TIMEOUT`, `TCP_CONGESTION` were added.

Modifié dans la version 3.6.5 : On Windows, `TCP_FASTOPEN`, `TCP_KEEPCNT` appear if run-time Windows supports.

`socket.AF_CAN`

`socket.PF_CAN`

SOL_CAN_***CAN_***

Many constants of these forms, documented in the Linux documentation, are also defined in the `socket` module.

Availability : Linux >= 2.6.25.

Nouveau dans la version 3.3.

`socket.CAN_BCM`

CAN_BCM_*

`CAN_BCM`, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the `socket` module.

Availability : Linux >= 2.6.25.

Nouveau dans la version 3.4.

`socket.CAN_RAW_FD_FRAMES`

Enables CAN FD support in a `CAN_RAW` socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you one must accept both CAN and CAN FD frames when reading from the socket.

This constant is documented in the Linux documentation.

Availability : Linux >= 3.6.

Nouveau dans la version 3.5.

`socket.AF_PACKET`

`socket.PF_PACKET`

PACKET_*

Many constants of these forms, documented in the Linux documentation, are also defined in the `socket` module.

Availability : Linux >= 2.2.

`socket.AF_RDS`

`socket.PF_RDS`

`socket.SOL_RDS`

RDS_*

Many constants of these forms, documented in the Linux documentation, are also defined in the `socket` module.

Availability : Linux >= 2.6.30.

Nouveau dans la version 3.3.

`socket.SIO_RCVALL`

`socket.SIO_KEEPA_LIVE_VALS`

`socket.SIO_LOOPBACK_FAST_PATH`

RCVALL_*

Constants for Windows' `WSAIocctl()`. The constants are used as arguments to the `ioctl()` method of socket objects.

Modifié dans la version 3.6 : `SIO_LOOPBACK_FAST_PATH` was added.

TIPC_*

TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

`socket.AF_ALG`

`socket.SOL_ALG`

ALG_*

Constants for Linux Kernel cryptography.

Availability : Linux >= 2.6.38.

Nouveau dans la version 3.6.

`socket.AF_LINK`

Availability : BSD, OSX.

Nouveau dans la version 3.4.

`socket.has_ipv6`

This constant contains a boolean value which indicates if IPv6 is supported on this platform.

`socket.BDADDR_ANY`

`socket.BDADDR_LOCAL`

These are string constants containing Bluetooth addresses with special meanings. For example, `BDADDR_ANY` can be used to indicate any address when specifying the binding socket with `BTPROTO_RFCOMM`.

`socket.HCI_FILTER`

`socket.HCI_TIME_STAMP`

`socket.HCI_DATA_DIR`

For use with `BTPROTO_HCI`. `HCI_FILTER` is not available for NetBSD or DragonFlyBSD. `HCI_TIME_STAMP` and `HCI_DATA_DIR` are not available for FreeBSD, NetBSD, or DragonFlyBSD.

Fonctions

Creating sockets

The following functions all create *socket objects*.

`socket.socket (family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)`

Create a new socket using the given address family, socket type and protocol number. The address family should be `AF_INET` (the default), `AF_INET6`, `AF_UNIX`, `AF_CAN`, `AF_PACKET`, or `AF_RDS`. The socket type should be `SOCK_STREAM` (the default), `SOCK_DGRAM`, `SOCK_RAW` or perhaps one of the other `SOCK_` constants. The protocol number is usually zero and may be omitted or in the case where the address family is `AF_CAN` the protocol should be one of `CAN_RAW` or `CAN_BCM`. If `fileno` is specified, the other arguments are ignored, causing the socket with the specified file descriptor to return. Unlike `socket.fromfd()`, `fileno` will return the same socket and not a duplicate. This may help close a detached socket using `socket.close()`.

Il n'est *pas possible d'hériter* du connecteur nouvellement créé.

Modifié dans la version 3.3 : The `AF_CAN` family was added. The `AF_RDS` family was added.

Modifié dans la version 3.4 : The `CAN_BCM` protocol was added.

Modifié dans la version 3.4 : The returned socket is now non-inheritable.

`socket.socketpair ([family[, type[, proto]]])`

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the `socket()` function above. The default family is `AF_UNIX` if defined on the platform; otherwise, the default is `AF_INET`.

The newly created sockets are *non-inheritable*.

Modifié dans la version 3.2 : The returned socket objects now support the whole socket API, rather than a subset.

Modifié dans la version 3.4 : The returned sockets are now non-inheritable.

Modifié dans la version 3.5 : Windows support added.

`socket.create_connection(address[, timeout[, source_address]])`

Connect to a TCP service listening on the Internet *address* (a 2-tuple (host, port)), and return the socket object. This is a higher-level function than `socket.connect()` : if *host* is a non-numeric hostname, it will try to resolve it for both `AF_INET` and `AF_INET6`, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional *timeout* parameter will set the timeout on the socket instance before attempting to connect. If no *timeout* is supplied, the global default timeout setting returned by `getdefaulttimeout()` is used.

If supplied, *source_address* must be a 2-tuple (host, port) for the socket to bind to as its source address before connecting. If host or port are "" or 0 respectively the OS default behavior will be used.

Modifié dans la version 3.2 : *source_address* was added.

`socket.fromfd(fd, family, type, proto=0)`

Duplicate the file descriptor *fd* (an integer as returned by a file object's `fileno()` method) and build a socket object from the result. Address family, socket type and protocol number are as for the `socket()` function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

Il n'est *pas possible d'hériter* du connecteur nouvellement créé.

Modifié dans la version 3.4 : The returned socket is now non-inheritable.

`socket.fromshare(data)`

Instantiate a socket from data obtained from the `socket.share()` method. The socket is assumed to be in blocking mode.

Disponibilité : Windows.

Nouveau dans la version 3.3.

`socket.SocketType`

This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

Autres fonctions

The `socket` module also offers various network-related services :

`socket.getaddrinfo(host, port, family=0, type=0, proto=0, flags=0)`

Translate the *host/port* argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. *host* is a domain name, a string representation of an IPv4/v6 address or None. *port* is a string service name such as 'http', a numeric port number or None. By passing None as the value of *host* and *port*, you can pass NULL to the underlying C API.

The *family*, *type* and *proto* arguments can be optionally specified in order to narrow the list of addresses returned. Passing zero as a value for each of these arguments selects the full range of results. The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. For example, `AI_NUMERICHOST` will disable domain name resolution and will raise an error if *host* is a domain name.

The function returns a list of 5-tuples with the following structure :

(family, type, proto, canonname, sockaddr)

In these tuples, *family*, *type*, *proto* are all integers and are meant to be passed to the `socket()` function. *canonname* will be a string representing the canonical name of the *host* if `AI_CANONNAME` is part of the *flags* argument ; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a (address, port) 2-tuple for `AF_INET`, a (address, port, flow info, scope id) 4-tuple for `AF_INET6`), and is meant to be passed to the `socket.connect()` method.

The following example fetches address information for a hypothetical TCP connection to `example.org` on port 80 (results may differ on your system if IPv6 isn't enabled) :

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
  6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
  6, '', ('93.184.216.34', 80))]
```

Modifié dans la version 3.2 : parameters can now be passed using keyword arguments.

`socket.getfqdn([name])`

Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by `gethostbyaddr()` is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available, the hostname as returned by `gethostname()` is returned.

`socket.gethostbyname(hostname)`

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as '100.50.200.5'. If the host name is an IPv4 address itself it is returned unchanged. See `gethostbyname_ex()` for a more complete interface. `gethostbyname()` does not support IPv6 name resolution, and `getaddrinfo()` should be used instead for IPv4/v6 dual stack support.

`socket.gethostbyname_ex(hostname)`

Translate a host name to IPv4 address format, extended interface. Return a triple (*hostname*, *aliaslist*, *ipaddrlist*) where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). `gethostbyname_ex()` does not support IPv6 name resolution, and `getaddrinfo()` should be used instead for IPv4/v6 dual stack support.

`socket.gethostname()`

Return a string containing the hostname of the machine where the Python interpreter is currently executing.

Note : `gethostname()` doesn't always return the fully qualified domain name ; use `getfqdn()` for that.

`socket.gethostbyaddr(ip_address)`

Return a triple (*hostname*, *aliaslist*, *ipaddrlist*) where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function `getfqdn()`. `gethostbyaddr()` supports both IPv4 and IPv6.

`socket.getnameinfo(sockaddr, flags)`

Translate a socket address *sockaddr* into a 2-tuple (*host*, *port*). Depending on the settings of *flags*, the result can contain a fully-qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number.

`socket.getprotobyne(protocolname)`

Translate an Internet protocol name (for example, 'icmp') to a constant suitable for passing as the (optional) third argument to the `socket()` function. This is usually only needed for sockets opened in « raw » mode (`SOCK_RAW`); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

`socket.getservbyname(servicename[, protocolname])`

Translate an Internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

`socket.getservbyport(port[, protocolname])`

Translate an Internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

`socket.ntohl(x)`

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the

same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

`socket.ntohs(x)`

Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

`socket.htonl(x)`

Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

`socket.htons(x)`

Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

`socket.inet_aton(ip_string)`

Convert an IPv4 address from dotted-quad string format (for example, “123.45.67.89”) to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary this function returns.

`inet_aton()` also accepts strings with less than three dots; see the Unix manual page `inet(3)` for details.

If the IPv4 address string passed to this function is invalid, `OSError` will be raised. Note that exactly what is valid depends on the underlying C implementation of `inet_aton()`.

`inet_aton()` does not support IPv6, and `inet_pton()` should be used instead for IPv4/v6 dual stack support.

`socket.inet_ntoa(packed_ip)`

Convert a 32-bit packed IPv4 address (a *bytes-like object* four bytes in length) to its standard dotted-quad string representation (for example, “123.45.67.89”). This is useful when conversing with a program that uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, `OSError` will be raised. `inet_ntoa()` does not support IPv6, and `inet_ntop()` should be used instead for IPv4/v6 dual stack support.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

`socket.inet_pton(address_family, ip_string)`

Convert an IP address from its family-specific string format to a packed, binary format. `inet_pton()` is useful when a library or network protocol calls for an object of type `struct in_addr` (similar to `inet_aton()`) or `struct in6_addr`.

Supported values for `address_family` are currently `AF_INET` and `AF_INET6`. If the IP address string `ip_string` is invalid, `OSError` will be raised. Note that exactly what is valid depends on both the value of `address_family` and the underlying implementation of `inet_pton()`.

Availability : Unix (maybe not all platforms), Windows.

Modifié dans la version 3.4 : Ajout de la gestion de Windows.

`socket.inet_ntop(address_family, packed_ip)`

Convert a packed IP address (a *bytes-like object* of some number of bytes) to its standard, family-specific string representation (for example, '7.10.0.5' or '5aef:2b::8'). `inet_ntop()` is useful when a library or network protocol returns an object of type `struct in_addr` (similar to `inet_ntoa()`) or `struct in6_addr`.

Supported values for `address_family` are currently `AF_INET` and `AF_INET6`. If the bytes object `packed_ip` is not the correct length for the specified address family, `ValueError` will be raised. `OSError` is raised for errors from the call to `inet_ntop()`.

Availability : Unix (maybe not all platforms), Windows.

Modifié dans la version 3.4 : Ajout de la gestion de Windows.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

`socket.CMSG_LEN (length)`

Return the total length, without trailing padding, of an ancillary data item with associated data of the given *length*. This value can often be used as the buffer size for `recvmsg()` to receive a single item of ancillary data, but [RFC 3542](#) requires portable applications to use `CMSG_SPACE()` and thus include space for padding, even when the item will be the last in the buffer. Raises `OverflowError` if *length* is outside the permissible range of values.

Availability : most Unix platforms, possibly others.

Nouveau dans la version 3.3.

`socket.CMSG_SPACE (length)`

Return the buffer size needed for `recvmsg()` to receive an ancillary data item with associated data of the given *length*, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the `CMSG_SPACE()` values for their associated data lengths. Raises `OverflowError` if *length* is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

Availability : most Unix platforms, possibly others.

Nouveau dans la version 3.3.

`socket.getdefaulttimeout ()`

Return the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

`socket.setdefaulttimeout (timeout)`

Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is `None`. See `settimeout()` for possible values and their respective meanings.

`socket.sethostname (name)`

Set the machine's hostname to *name*. This will raise an `OSError` if you don't have enough rights.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`socket.if_nameindex ()`

Return a list of network interface information (index int, name string) tuples. `OSError` if the system call fails.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`socket.if_nameindex (if_name)`

Return a network interface index number corresponding to an interface name. `OSError` if no interface with the given name exists.

Disponibilité : Unix.

Nouveau dans la version 3.3.

`socket.if_indextoname (if_index)`

Return a network interface name corresponding to an interface index number. `OSError` if no interface with the given index exists.

Disponibilité : Unix.

Nouveau dans la version 3.3.

18.1.3 Socket Objects

Socket objects have the following methods. Except for `makefile()`, these correspond to Unix system calls applicable to sockets.

Modifié dans la version 3.2 : Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling `close()`.

`socket.accept()`

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where `conn` is a *new* socket object usable to send and receive data on the connection, and `address` is the address bound to the socket on the other end of the connection.

Il n'est *pas possible d'hériter* du connecteur nouvellement créé.

Modifié dans la version 3.4 : The socket is now non-inheritable.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

`socket.bind(address)`

Bind the socket to `address`. The socket must not already be bound. (The format of `address` depends on the address family — see above.)

`socket.close()`

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from `makefile()` are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to `close()` them explicitly, or to use a `with` statement around them.

Modifié dans la version 3.6 : `OSError` is now raised if an error occurs when the underlying `close()` call is made.

Note : `close()` releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call `shutdown()` before `close()`.

`socket.connect(address)`

Connect to a remote socket at `address`. (The format of `address` depends on the address family — see above.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a `socket.timeout` on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an `InterruptedError` exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

Modifié dans la version 3.5 : The method now waits until the connection completes instead of raising an `InterruptedError` exception if the connection is interrupted by a signal, the signal handler doesn't raise an exception and the socket is blocking or has a timeout (see the [PEP 475](#) for the rationale).

`socket.connect_ex(address)`

Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level `connect()` call (other problems, such as « host not found, » can still raise exceptions). The error indicator is 0 if the operation succeeded, otherwise the value of the `errno` variable. This is useful to support, for example, asynchronous connects.

`socket.detach()`

Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

Nouveau dans la version 3.2.

`socket.dup()`

Duplicate the socket.

Il n'est *pas possible d'hériter* du connecteur nouvellement créé.

Modifié dans la version 3.4 : The socket is now non-inheritable.

`socket.fileno()`

Return the socket's file descriptor (a small integer), or -1 on failure. This is useful with `select.select()`.

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as `os.fdopen()`). Unix does not have this limitation.

`socket.get_inheritable()`

Get the *inheritable flag* of the socket's file descriptor or socket's handle : `True` if the socket can be inherited in child processes, `False` if it cannot.

Nouveau dans la version 3.4.

`socket.getpeername()`

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

`socket.getsockname()`

Return the socket's own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

`socket.getsockopt(level, optname[, buflen])`

Return the value of the given socket option (see the Unix man page `getsockopt(2)`). The needed symbolic constants (`SO_*` etc.) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module `struct` for a way to decode C structures encoded as byte strings).

`socket.gettimeout()`

Return the timeout in seconds (float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to `setblocking()` or `settimeout()`.

`socket.ioctl(control, option)`

Platform Windows

The `ioctl()` method is a limited interface to the `WSAIoctl` system interface. Please refer to the [Win32 documentation](#) for more information.

On other platforms, the generic `fcntl.fcntl()` and `fcntl.ioctl()` functions may be used; they accept a socket object as their first argument.

Currently only the following control codes are supported : `SIO_RCVALL`, `SIO_KEEPAIVE_VALS`, and `SIO_LOOPBACK_FAST_PATH`.

Modifié dans la version 3.6 : `SIO_LOOPBACK_FAST_PATH` was added.

`socket.listen([backlog])`

Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

Modifié dans la version 3.5 : The *backlog* parameter is now optional.

`socket.makefile(mode='r', buffering=None, *, encoding=None, errors=None, newline=None)`

Return a *file object* associated with the socket. The exact returned type depends on the arguments given to `makefile()`. These arguments are interpreted the same way as by the built-in `open()` function, except the only supported *mode* values are 'r' (default), 'w' and 'b'.

The socket must be in blocking mode; it can have a timeout, but the file object's internal buffer may end up in an inconsistent state if a timeout occurs.

Closing the file object returned by `makefile()` won't close the original socket unless all other file objects have been closed and `socket.close()` has been called on the socket object.

Note : On Windows, the file-like object created by `makefile()` cannot be used where a file object with a file descriptor is expected, such as the stream arguments of `subprocess.Popen()`.

`socket.recv(bufsize[, flags])`

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by `bufsize`. See the Unix manual page `recv(2)` for the meaning of the optional argument `flags`; it defaults to zero.

Note : For best match with hardware and network realities, the value of `bufsize` should be a relatively small power of 2, for example, 4096.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

`socket.recvfrom(bufsize[, flags])`

Receive data from the socket. The return value is a pair (`bytes`, `address`) where `bytes` is a bytes object representing the data received and `address` is the address of the socket sending the data. See the Unix manual page `recv(2)` for the meaning of the optional argument `flags`; it defaults to zero. (The format of `address` depends on the address family — see above.)

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

`socket.recvmsg(bufsize[, ancbufsize[, flags]])`

Receive normal data (up to `bufsize` bytes) and ancillary data from the socket. The `ancbufsize` argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using `CMSG_SPACE()` or `CMSG_LEN()`, and items which do not fit into the buffer might be truncated or discarded. The `flags` argument defaults to 0 and has the same meaning as for `recv()`.

The return value is a 4-tuple : (`data`, `ancdata`, `msg_flags`, `address`). The `data` item is a `bytes` object holding the non-ancillary data received. The `ancdata` item is a list of zero or more tuples (`cmsg_level`, `cmsg_type`, `cmsg_data`) representing the ancillary data (control messages) received : `cmsg_level` and `cmsg_type` are integers specifying the protocol level and protocol-specific type respectively, and `cmsg_data` is a `bytes` object holding the associated data. The `msg_flags` item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, `address` is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, `sendmsg()` and `recvmsg()` can be used to pass file descriptors between processes over an `AF_UNIX` socket. When this facility is used (it is often restricted to `SOCK_STREAM` sockets), `recvmsg()` will return, in its ancillary data, items of the form (`socket.SOL_SOCKET`, `socket.SCM_RIGHTS`, `fds`), where `fds` is a `bytes` object representing the new file descriptors as a binary array of the native C `int` type. If `recvmsg()` raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, `recvmsg()` will issue a `RuntimeWarning`, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the `SCM_RIGHTS` mechanism, the following function will receive up to `maxfds` file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also `sendmsg()`.

```

import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")    # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.
    ↪itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if (cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS):
            # Append data, ignoring any truncated integers at the end.
            fds.fromstring(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.
    ↪itemsize)])
    return msg, list(fds)

```

Availability : most Unix platforms, possibly others.

Nouveau dans la version 3.3.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une *InterruptedError* (voir la [PEP 475](#) à propos du raisonnement).

`socket.recvmsg_into(buffers[, ancbufsize[, flags]])`

Receive normal data and ancillary data from the socket, behaving as `recvmsg()` would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The *buffers* argument must be an iterable of objects that export writable buffers (e.g. *bytearray* objects); these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancbufsize* and *flags* arguments have the same meaning as for `recvmsg()`.

The return value is a 4-tuple : (*nbytes*, *ancdata*, *msg_flags*, *address*), where *nbytes* is the total number of bytes of non-ancillary data written into the buffers, and *ancdata*, *msg_flags* and *address* are the same as for `recvmsg()`.

Exemple :

```

>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'-----')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]

```

Availability : most Unix platforms, possibly others.

Nouveau dans la version 3.3.

`socket.recvfrom_into(buffer[, nbytes[, flags]])`

Receive data from the socket, writing it into *buffer* instead of creating a new bytestring. The return value is a pair (*nbytes*, *address*) where *nbytes* is the number of bytes received and *address* is the address of the socket sending the data. See the Unix manual page `recv(2)` for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

`socket.recv_into(buffer[, nbytes[, flags]])`

Receive up to *nbytes* bytes from the socket, storing the data into a buffer rather than creating a new bytestring. If *nbytes* is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page `recv(2)` for the meaning of the optional argument *flags*; it defaults to zero.

`socket.send(bytes[, flags])`

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for `recv()` above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the `socket-howto`.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

`socket.sendall(bytes[, flags])`

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for `recv()` above. Unlike `send()`, this method continues to send data from *bytes* until either all data has been sent or an error occurs. `None` is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

Modifié dans la version 3.5 : The socket timeout is no more reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

`socket.sendto(bytes, address)`

`socket.sendto(bytes, flags, address)`

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for `recv()` above. Return the number of bytes sent. (The format of *address* depends on the address family — see above.)

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

`socket.sendmsg(bufers[, ancdata[, flags[, address]]])`

Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The *bufers* argument specifies the non-ancillary data as an iterable of *bytes-like objects* (e.g. `bytes` objects); the operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancdata* argument specifies the ancillary data (control messages) as an iterable of zero or more tuples (`cmsg_level`, `cmsg_type`, `cmsg_data`), where *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a bytes-like object holding the associated data. Note that some systems (in particular, systems without `MSG_SPACE()`) might support sending only one control message per call. The *flags* argument defaults to 0 and has the same meaning as for `send()`. If *address* is supplied and not `None`, it sets a destination address for the message. The return value is the number of bytes of non-ancillary data sent.

The following function sends the list of file descriptors *fds* over an `AF_UNIX` socket, on systems which support the `SCM_RIGHTS` mechanism. See also `recvmsg()`.

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.
    ↪array("i", fds))])
```

Availability : most Unix platforms, possibly others.

Nouveau dans la version 3.3.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la [PEP 475](#) à propos du raisonnement).

`socket.sendmsg_afalg([msg], *, op[, iv[, assoclen[, flags]]])`

Specialized version of `sendmsg()` for `AF_ALG` socket. Set mode, IV, AEAD associated data length and flags for `AF_ALG` socket.

Availability : Linux >= 2.6.38

Nouveau dans la version 3.6.

`socket.sendfile(file, offset=0, count=None)`

Send a file until EOF is reached by using high-performance `os.sendfile` and return the total number of bytes which were sent. `file` must be a regular file object opened in binary mode. If `os.sendfile` is not available (e.g. Windows) or `file` is not a regular file `send()` will be used instead. `offset` tells from where to start reading the file. If specified, `count` is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is updated on return or also in case of error in which case `file.tell()` can be used to figure out the number of bytes which were sent. The socket must be of `SOCK_STREAM` type. Non-blocking sockets are not supported.

Nouveau dans la version 3.5.

`socket.set_inheritable(inheritable)`

Set the *inheritable flag* of the socket's file descriptor or socket's handle.

Nouveau dans la version 3.4.

`socket.setblocking(flag)`

Set blocking or non-blocking mode of the socket : if `flag` is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain `settimeout()` calls :

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0.0)`

`socket.settimeout(value)`

Set a timeout on blocking socket operations. The `value` argument can be a nonnegative floating point number expressing seconds, or `None`. If a non-zero value is given, subsequent socket operations will raise a *timeout* exception if the timeout period `value` has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If `None` is given, the socket is put in blocking mode.

For further information, please consult the *notes on socket timeouts*.

`socket.setsockopt(level, optname, value : int)`

`socket.setsockopt(level, optname, value : buffer)`

`socket.setsockopt(level, optname, None, optlen : int)`

Set the value of the given socket option (see the Unix manual page `setsockopt(2)`). The needed symbolic constants are defined in the `socket` module (`SO_*` etc.). The value can be an integer, `None` or a *bytes-like object* representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module `struct` for a way to encode C structures as bytestrings). When value is set to `None`, `optlen` argument is required. It's equivalent to call `setsockopt` C function with `optval=NULL` and `optlen=optlen`.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

Modifié dans la version 3.6 : `setsockopt(level, optname, None, optlen : int)` form added.

`socket.shutdown(how)`

Shut down one or both halves of the connection. If `how` is `SHUT_RD`, further receives are disallowed. If `how` is `SHUT_WR`, further sends are disallowed. If `how` is `SHUT_RDWR`, further sends and receives are disallowed.

`socket.share(process_id)`

Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with `process_id`. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using `fromshare()`. Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

Disponibilité : Windows.

Nouveau dans la version 3.3.

Note that there are no methods `read()` or `write()`; use `recv()` and `send()` without *flags* argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the `socket` constructor.

`socket.family`

The socket family.

`socket.type`

The socket type.

`socket.proto`

The socket protocol.

18.1.4 Notes on socket timeouts

A socket object can be in one of three modes : blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling `setdefaulttimeout()`.

- In *blocking mode*, operations block until complete or the system returns an error (such as connection timed out).
- In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately : functions from the `select` can be used to know when and whether a socket is available for reading or writing.
- In *timeout mode*, operations fail if they cannot be completed within the timeout specified for the socket (they raise a `timeout` exception) or if the system returns an error.

Note : At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the `fileno()` of a socket.

Timeouts and the `connect` method

The `connect()` operation is also subject to the timeout setting, and in general it is recommended to call `settimeout()` before calling `connect()` or pass a timeout parameter to `create_connection()`. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

Timeouts and the `accept` method

If `getdefaulttimeout()` is not `None`, sockets returned by the `accept()` method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket :

- if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by `accept()` is in *blocking mode*;
- if the listening socket is in *non-blocking mode*, whether the socket returned by `accept()` is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

18.1.5 Exemple

Here are four minimal example programs using the TCP/IP protocol : a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket()`, `bind()`, `listen()`, `accept()` (possibly repeating the `accept()` to service more than one client), while a client only needs the sequence `socket()`, `connect()`. Also note that the server does not `sendall()/recv()` on the socket it is listening on but on the new socket returned by `accept()`.

The first two examples support IPv4 only.

```
# Echo server program
import socket

HOST = ''          # Symbolic name meaning all available interfaces
PORT = 50007       # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl' # The remote host
PORT = 50007           # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to the all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None        # Symbolic name meaning all available interfaces
PORT = 50007       # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
```

(suite sur la page suivante)

(suite de la page précédente)

```

try:
    s.bind(sa)
    s.listen(1)
except OSError as msg:
    s.close()
    s = None
    continue
break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)

```

```

# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'      # The remote host
PORT = 50007                # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))

```

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface :

```

import socket

# the public network interface
HOST = socket.gethostname(socket.gethostname())

```

(suite sur la page suivante)

(suite de la page précédente)

```
# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The last example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with :

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

After binding (`CAN_RAW`) or connecting (`CAN_BCM`) the socket, you can use the `socket.send()`, and the `socket.recv()` operations (and their counterparts) on the socket object as usual.

This example might require special privileges :

```
import socket
import struct

# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])

# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
```

(suite sur la page suivante)

(suite de la page précédente)

```
print('Error sending CAN frame')

try:
    s.send(build_can_frame(0x01, b'\x01\x02\x03'))
except OSError:
    print('Error sending CAN frame')
```

Running an example several times with too small delay between executions, could lead to this error :

```
OSError: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can't be immediately reused.

There is a `socket` flag to set, in order to prevent this, `socket.SO_REUSEADDR` :

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the `SO_REUSEADDR` flag tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.

Voir aussi :

For an introduction to socket programming (in C), see the following papers :

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al,

both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1 :7 and PS1 :8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages ; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to [RFC 3493](#) titled Basic Socket Interface Extensions for IPv6.

18.2 ssl — Emballage TLS/SSL pour les objets connecteurs

Code source : [Lib/ssl.py](#)

Ce module fournit un accès aux fonctions de chiffrement et d'authentification entre pairs : « *Transport Layer Security* » (souvent appelé « *Secure Sockets Layer* ») pour les connecteurs réseau, côté client et côté serveur. Ce module utilise la bibliothèque OpenSSL. Il est disponible sur tous les systèmes Unix modernes, Windows, Mac OS X et probablement sur d'autres plates-formes, à condition qu'OpenSSL soit installé sur cette plate-forme.

Note : Certains comportements peuvent dépendre de la plate-forme, car des appels sont passés aux API de connexions du système d'exploitation. La version installée de OpenSSL peut également entraîner des variations de comportement. Par exemple, TLSv1.1 et TLSv1.2 sont livrés avec la version 1.0.1 de OpenSSL.

Avertissement : N'utilisez pas ce module sans lire [Security considerations](#). Cela pourrait créer un faux sentiment de sécurité, car les paramètres par défaut du module `ssl` ne sont pas nécessairement appropriés pour votre application.

Cette section documente les objets et les fonctions du module `ssl`. Pour des informations plus générales sur TLS, SSL et les certificats, le lecteur est prié de se référer aux documents de la section « Voir Aussi » au bas de cette page.

Ce module fournit une classe `ssl.SSLSocket`, dérivée du type `socket.socket`, ainsi qu'une enveloppe (*wrapper* en anglais) qui mime un connecteur qui chiffre et déchiffre les données passant sur le connecteur utilisant SSL. Il prend en charge des méthodes supplémentaires telles que `getpeercert()` qui récupère le certificat de l'autre coté de la connexion et `cipher()`, qui récupère le chiffrement utilisé pour la connexion.

Pour les applications plus sophistiquées, la classe `ssl.SSLContext` facilite la gestion des paramètres et des certificats, qui peuvent ensuite être hérités par les connecteurs SSL créés via la méthode `SSLContext.wrap_socket()`.

Modifié dans la version 3.5.3 : Mise à jour pour prendre en charge la liaison avec OpenSSL 1.1.0

Modifié dans la version 3.6 : OpenSSL 0.9.8, 1.0.0 et 1.0.1 sont obsolètes et ne sont plus prises en charge. Dans l'avenir, le module `ssl` nécessitera au minimum OpenSSL 1.0.2 ou 1.1.0.

18.2.1 Fonctions, constantes et exceptions

exception `ssl.SSLError`

Levée pour signaler une erreur de l'implémentation SSL sous-jacente (actuellement fournie par la bibliothèque OpenSSL). Cela signifie qu'un problème est apparu dans la couche d'authentification et de chiffrement de niveau supérieur qui s'appuie sur la connexion réseau sous-jacente. Cette erreur est un sous-type de `OSError`. Le code d'erreur et le message des instances de `SSLError` sont fournis par la bibliothèque OpenSSL.

Modifié dans la version 3.3 : `SSLError` était un sous-type de `socket.error`.

library

Une chaîne de caractères mnémonique désignant le sous-module OpenSSL dans lequel l'erreur s'est produite, telle que `SSL`, `PEM` ou `X509`. L'étendue des valeurs possibles dépend de la version d'OpenSSL.

Nouveau dans la version 3.3.

reason

A string mnemonic designating the reason this error occurred, for example `CERTIFICATE_VERIFY_FAILED`. The range of possible values depends on the OpenSSL version.

Nouveau dans la version 3.3.

exception `ssl.SSLZeroReturnError`

A subclass of `SSLError` raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn't mean that the underlying transport (read TCP) has been closed.

Nouveau dans la version 3.3.

exception `ssl.SSLWantReadError`

Sous-classe de `SSLError` levée par un connecteur *SSL non bloquant* lors d'une tentative de lecture ou d'écriture de données, alors que davantage de données doivent être reçues sur la couche TCP sous-jacente avant que la demande puisse être satisfaite.

Nouveau dans la version 3.3.

exception `ssl.SSLWantWriteError`

A subclass of `SSLError` raised by a *non-blocking SSL socket* when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

Nouveau dans la version 3.3.

exception `ssl.SSLSyscallError`

A subclass of `SSLError` raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

Nouveau dans la version 3.3.

exception `ssl.SSL_EOFError`

A subclass of `SSLError` raised when the SSL connection has been terminated abruptly. Generally, you shouldn't try to reuse the underlying transport when this error is encountered.

Nouveau dans la version 3.3.

exception `ssl.CertificateError`

Raised to signal an error with a certificate (such as mismatching hostname). Certificate errors detected by OpenSSL, though, raise an `SSL.Error`.

Création de connecteurs

The following function allows for standalone socket creation. Starting from Python 3.2, it can be more flexible to use `SSLContext.wrap_socket()` instead.

`ssl.wrap_socket(sock, keyfile=None, certfile=None, server_side=False, cert_reqs=CERT_NONE, ssl_version={see docs}, ca_certs=None, do_handshake_on_connect=True, suppress_ragged_eofs=True, ciphers=None)`

Takes an instance `sock` of `socket.socket`, and returns an instance of `ssl.SSLSocket`, a subtype of `socket.socket`, which wraps the underlying socket in an SSL context. `sock` must be a `SOCK_STREAM` socket; other socket types are unsupported.

For client-side sockets, the context construction is lazy; if the underlying socket isn't connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. `wrap_socket()` may raise `SSL.Error`.

The `keyfile` and `certfile` parameters specify optional files which contain a certificate to be used to identify the local side of the connection. See the discussion of [Certificates](#) for more information on how the certificate is stored in the `certfile`.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

The parameter `cert_reqs` specifies whether a certificate is required from the other side of the connection, and whether it will be validated if provided. It must be one of the three values `CERT_NONE` (certificates ignored), `CERT_OPTIONAL` (not required, but validated if provided), or `CERT_REQUIRED` (required and validated). If the value of this parameter is not `CERT_NONE`, then the `ca_certs` parameter must point to a file of CA certificates.

The `ca_certs` file contains a set of concatenated « certification authority » certificates, which are used to validate certificates passed from the other end of the connection. See the discussion of [Certificates](#) for more information about how to arrange the certificates in this file.

The parameter `ssl_version` specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server's choice. Most of the versions are not interoperable with the other versions. If not specified, the default is `PROTOCOL_TLS`; it provides the most compatibility with other versions.

Here's a table showing which versions in a client (down the side) can connect to which versions in a server (along the top) :

<i>client / server</i>	SSLv2	SSLv3	TLS ³	TLSv1	TLSv1.1	TLSv1.2
SSLv2	oui	non	no ¹	non	non	non
SSLv3	non	oui	no ²	non	non	non
TLS (SSLv23) ³	no ¹	no ²	oui	oui	oui	oui
TLSv1	non	non	oui	oui	non	non
TLSv1.1	non	non	oui	non	oui	non
TLSv1.2	non	non	oui	non	non	oui

3. TLS 1.3 protocol will be available with `PROTOCOL_TLS` in OpenSSL >= 1.1.1. There is no dedicated `PROTOCOL` constant for just TLS 1.3.

1. `SSLContext` disables SSLv2 with `OP_NO_SSLv2` by default.

2. `SSLContext` disables SSLv3 with `OP_NO_SSLv3` by default.

Notes

Note : Which connections succeed will vary depending on the version of OpenSSL. For example, before OpenSSL 1.0.0, an SSLv23 client would always attempt SSLv2 connections.

The *ciphers* parameter sets the available ciphers for this SSL object. It should be a string in the [OpenSSL cipher list format](#).

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a `socket.connect()`, or whether the application program will call it explicitly, by invoking the `SSLSocket.do_handshake()` method. Calling `SSLSocket.do_handshake()` explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the `SSLSocket.recv()` method should signal unexpected EOF from the other end of the connection. If specified as *True* (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if *False*, it will raise the exceptions back to the caller.

Modifié dans la version 3.2 : New optional argument *ciphers*.

Création de contexte

Une fonction utilitaire permettant de créer facilement des objets `SSLContext` pour des usages classiques.

`ssl.create_default_context(purpose=Purpose.SERVER_AUTH, cafile=None, capath=None, cadata=None)`

Renvoie un nouvel objet `SSLContext`. Le paramètre *purpose* permet de choisir parmi un ensemble de paramètres par défaut en fonction de l'usage souhaité. Les paramètres sont choisis par le module `ssl` et représentent généralement un niveau de sécurité supérieur à celui utilisé lorsque vous appelez directement le constructeur `SSLContext`. *cafile*, *capath*, *cadata* représentent des certificats d'autorité de certification facultatifs approuvés pour la vérification de certificats, comme dans `SSLContext.load_verify_locations()`. Si les trois sont à *None*, cette fonction peut choisir de faire confiance aux certificats d'autorité de certification par défaut du système.

Les paramètres sont : `PROTOCOL_TLS`, `OP_NO_SSLv2` et `OP_NO_SSLv3` avec des algorithmes de chiffrement de grande robustesse, n'utilisant pas RC4 et n'utilisant pas les suites cryptographiques sans authentification. Passer `SERVER_AUTH` en tant que *purpose* définit *verify_mode* sur `CERT_REQUIRED` et charge les certificats de l'autorité de certification (lorsqu'au moins un des paramètres *cafile*, *capath* ou *cadata* est renseigné) ou utilise `SSLContext.load_default_certs()` pour charger les certificats des autorités de certification par défaut.

Note : Le protocole, les options, l'algorithme de chiffrement et d'autres paramètres peuvent changer pour des valeurs plus restrictives à tout moment sans avertissement préalable. Les valeurs représentent un juste équilibre entre compatibilité et sécurité.

Si votre application nécessite des paramètres spécifiques, vous devez créer une classe `SSLContext` et appliquer les paramètres vous-même.

Note : Si vous constatez que, lorsque certains clients ou serveurs plus anciens tentent de se connecter avec une classe `SSLContext` créée par cette fonction, une erreur indiquant « *Protocol or cipher suite mismatch* » (« Non concordance de protocole ou d'algorithme de chiffrement ») est détectée, il se peut qu'ils ne prennent en charge que SSL 3.0 que cette fonction exclut en utilisant `OP_NO_SSLv3`. SSL3.0 est notoirement considéré comme [totalement déficient](#). Si vous souhaitez toujours continuer à utiliser cette fonction tout en autorisant les connexions SSL 3.0, vous pouvez les réactiver à l'aide de :

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options |= ~ssl.OP_NO_SSLv3
```

Nouveau dans la version 3.4.

Modifié dans la version 3.4.4 : RC4 a été supprimé de la liste des algorithmes de chiffrement par défaut.

Modifié dans la version 3.6 : *ChaCha20/Poly1305* a été ajouté à la liste des algorithmes de chiffrement par défaut.

3DES a été supprimé de la liste des algorithmes de chiffrement par défaut.

Random generation

`ssl.RAND_bytes(num)`

Return *num* cryptographically strong pseudo-random bytes. Raises an *SSLError* if the PRNG has not been seeded with enough data or if the operation is not supported by the current RAND method. `RAND_status()` can be used to check the status of the PRNG and `RAND_add()` can be used to seed the PRNG.

For almost all applications `os.urandom()` is preferable.

Read the Wikipedia article, [Cryptographically secure pseudorandom number generator \(CSPRNG\)](#), to get the requirements of a cryptographically generator.

Nouveau dans la version 3.3.

`ssl.RAND_pseudo_bytes(num)`

Return (bytes, is_cryptographic) : bytes are *num* pseudo-random bytes, is_cryptographic is True if the bytes generated are cryptographically strong. Raises an *SSLError* if the operation is not supported by the current RAND method.

Generated pseudo-random byte sequences will be unique if they are of sufficient length, but are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

For almost all applications `os.urandom()` is preferable.

Nouveau dans la version 3.3.

Obsolète depuis la version 3.6 : OpenSSL has deprecated `ssl.RAND_pseudo_bytes()`, use `ssl.RAND_bytes()` instead.

`ssl.RAND_status()`

Return True if the SSL pseudo-random number generator has been seeded with “enough” randomness, and False otherwise. You can use `ssl.RAND_egd()` and `ssl.RAND_add()` to increase the randomness of the pseudo-random number generator.

`ssl.RAND_egd(path)`

If you are running an entropy-gathering daemon (EGD) somewhere, and *path* is the pathname of a socket connection open to it, this will read 256 bytes of randomness from the socket, and add it to the SSL pseudo-random number generator to increase the security of generated secret keys. This is typically only necessary on systems without better sources of randomness.

See <http://egd.sourceforge.net/> or <http://prngd.sourceforge.net/> for sources of entropy-gathering daemons.

Availability : not available with LibreSSL and OpenSSL > 1.1.0

`ssl.RAND_add(bytes, entropy)`

Mix the given *bytes* into the SSL pseudo-random number generator. The parameter *entropy* (a float) is a lower bound on the entropy contained in string (so you can always use 0.0). See [RFC 1750](#) for more information on sources of entropy.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

Certificate handling

`ssl.match_hostname(cert, hostname)`

Verify that *cert* (in decoded format as returned by `SSLSocket.getpeercert()`) matches the given *hostname*. The rules applied are those for checking the identity of HTTPS servers as outlined in [RFC 2818](#), [RFC 5280](#) and [RFC 6125](#). In addition to HTTPS, this function should be suitable for checking the identity of servers in various SSL-based protocols such as FTPS, IMAPS, POPS and others.

`CertificateError` is raised on failure. On success, the function returns nothing :

```
>>> cert = {'subject': (('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

Nouveau dans la version 3.2.

Modifié dans la version 3.3.3 : The function now follows [RFC 6125](#), section 6.4.3 and does neither match multiple wildcards (e.g. `*.*.com` or `*a*.example.org`) nor a wildcard inside an internationalized domain names (IDN) fragment. IDN A-labels such as `www*.xn--python-kva.org` are still supported, but `x*.python.org` no longer matches `xn--tda.python.org`.

Modifié dans la version 3.5 : Matching of IP addresses, when present in the `subjectAltName` field of the certificate, is now supported.

`ssl.cert_time_to_seconds(cert_time)`

Return the time in seconds since the Epoch, given the *cert_time* string representing the « notBefore » or « notAfter » date from a certificate in "`%b %d %H:%M:%S %Y %Z`" `strptime` format (C locale).

Here's an example :

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

« notBefore » or « notAfter » dates must use GMT ([RFC 5280](#)).

Modifié dans la version 3.5 : Interpret the input time as a time in UTC as specified by “GMT” timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

`ssl.get_server_certificate(addr, ssl_version=PROTOCOL_TLS, ca_certs=None)`

Given the address *addr* of an SSL-protected server, as a (*hostname*, *port-number*) pair, fetches the server's certificate, and returns it as a PEM-encoded string. If *ssl_version* is specified, uses that version of the SSL protocol to attempt to connect to the server. If *ca_certs* is specified, it should be a file containing a list of root certificates, the same format as used for the same parameter in `wrap_socket()`. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails.

Modifié dans la version 3.3 : This function is now IPv6-compatible.

Modifié dans la version 3.5 : The default *ssl_version* is changed from `PROTOCOL_SSLv3` to `PROTOCOL_TLS` for maximum compatibility with modern servers.

`ssl.DER_cert_to_PEM_cert(DER_cert_bytes)`

Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

`ssl.PEM_cert_to_DER_cert(PEM_cert_string)`

Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

ssl.get_default_verify_paths()

Returns a named tuple with paths to OpenSSL's default cafile and capath. The paths are the same as used by `SSLContext.set_default_verify_paths()`. The return value is a *named tuple* `DefaultVerifyPaths`:

- `cafile` - resolved path to cafile or `None` if the file doesn't exist,
- `capath` - resolved path to capath or `None` if the directory doesn't exist,
- `openssl_cafile_env` - OpenSSL's environment key that points to a cafile,
- `openssl_cafile` - hard coded path to a cafile,
- `openssl_capath_env` - OpenSSL's environment key that points to a capath,
- `openssl_capath` - hard coded path to a capath directory

Availability : LibreSSL ignores the environment vars `openssl_cafile_env` and `openssl_capath_env`
Nouveau dans la version 3.4.

ssl.enum_certificates(store_name)

Retrieve certificates from Windows' system cert store. `store_name` may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (`cert_bytes`, `encoding_type`, `trust`) tuples. The `encoding_type` specifies the encoding of `cert_bytes`. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data. `trust` specifies the purpose of the certificate as a set of OIDS or exactly `True` if the certificate is trustworthy for all purposes.

Exemple :

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

Disponibilité : Windows.

Nouveau dans la version 3.4.

ssl.enum_crls(store_name)

Retrieve CRLs from Windows' system cert store. `store_name` may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (`cert_bytes`, `encoding_type`, `trust`) tuples. The `encoding_type` specifies the encoding of `cert_bytes`. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data.

Disponibilité : Windows.

Nouveau dans la version 3.4.

Constantes

All constants are now *enum.IntEnum* or *enum.IntFlag* collections.

Nouveau dans la version 3.6.

ssl.CERT_NONE

Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`. Except for `PROTOCOL_TLS_CLIENT`, it is the default mode. With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

In server mode, no certificate is requested from the client, so the client does not send any for client cert authentication.

See the discussion of *Security considerations* below.

ssl.CERT_OPTIONAL

Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`. In client mode, `CERT_OPTIONAL` has the same meaning as `CERT_REQUIRED`. It is recommended to use `CERT_REQUIRED` for client-side sockets instead.

In server mode, a client certificate request is sent to the client. The client may either ignore the request or send a certificate in order to perform TLS client cert authentication. If the client chooses to send a certificate, it is verified. Any verification error immediately aborts the TLS handshake.

Use of this setting requires a valid set of CA certificates to be passed, either to `SSLContext.load_verify_locations()` or as a value of the `ca_certs` parameter to `wrap_socket()`.

`ssl.CERT_REQUIRED`

Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`. In this mode, certificates are required from the other side of the socket connection; an `SSLError` will be raised if no certificate is provided, or if its validation fails. This mode is **not** sufficient to verify a certificate in client mode as it does not match hostnames. `check_hostname` must be enabled as well to verify the authenticity of a cert. `PROTOCOL_TLS_CLIENT` uses `CERT_REQUIRED` and enables `check_hostname` by default.

With server socket, this mode provides mandatory TLS client cert authentication. A client certificate request is sent to the client and the client must provide a valid and trusted certificate.

Use of this setting requires a valid set of CA certificates to be passed, either to `SSLContext.load_verify_locations()` or as a value of the `ca_certs` parameter to `wrap_socket()`.

`class ssl.VerifyMode`

`enum.IntEnum` collection of `CERT_*` constants.

Nouveau dans la version 3.6.

`ssl.VERIFY_DEFAULT`

Possible value for `SSLContext.verify_flags`. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.

Nouveau dans la version 3.4.

`ssl.VERIFY_CRL_CHECK_LEAF`

Possible value for `SSLContext.verify_flags`. In this mode, only the peer cert is checked but none of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper has been loaded `SSLContext.load_verify_locations`, validation will fail.

Nouveau dans la version 3.4.

`ssl.VERIFY_CRL_CHECK_CHAIN`

Possible value for `SSLContext.verify_flags`. In this mode, CRLs of all certificates in the peer cert chain are checked.

Nouveau dans la version 3.4.

`ssl.VERIFY_X509_STRICT`

Possible value for `SSLContext.verify_flags` to disable workarounds for broken X.509 certificates.

Nouveau dans la version 3.4.

`ssl.VERIFY_X509_TRUSTED_FIRST`

Possible value for `SSLContext.verify_flags`. It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

Nouveau dans la version 3.4.4.

`class ssl.VerifyFlags`

`enum.IntFlag` collection of `VERIFY_*` constants.

Nouveau dans la version 3.6.

`ssl.PROTOCOL_TLS`

Selects the highest protocol version that both the client and server support. Despite the name, this option can select both « SSL » and « TLS » protocols.

Nouveau dans la version 3.6.

`ssl.PROTOCOL_TLS_CLIENT`

Auto-negotiate the highest protocol version like `PROTOCOL_TLS`, but only support client-side `SSLSocket` connections. The protocol enables `CERT_REQUIRED` and `check_hostname` by default.

Nouveau dans la version 3.6.

ssl.PROTOCOL_TLS_SERVER

Auto-negotiate the highest protocol version like *PROTOCOL_TLS*, but only support server-side *SSLSocket* connections.

Nouveau dans la version 3.6.

ssl.PROTOCOL_SSLv23

Alias for data :*PROTOCOL_TLS*.

Obsolète depuis la version 3.6 : Use *PROTOCOL_TLS* instead.

ssl.PROTOCOL_SSLv2

Selects SSL version 2 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the *OPENSSL_NO_SSL2* flag.

Avertissement : SSL version 2 is insecure. Its use is highly discouraged.

Obsolète depuis la version 3.6 : OpenSSL has removed support for SSLv2.

ssl.PROTOCOL_SSLv3

Selects SSL version 3 as the channel encryption protocol.

This protocol is not be available if OpenSSL is compiled with the *OPENSSL_NO_SSLv3* flag.

Avertissement : SSL version 3 is insecure. Its use is highly discouraged.

Obsolète depuis la version 3.6 : OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.PROTOCOL_TLSv1

Selects TLS version 1.0 as the channel encryption protocol.

Obsolète depuis la version 3.6 : OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.PROTOCOL_TLSv1_1

Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Nouveau dans la version 3.4.

Obsolète depuis la version 3.6 : OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.PROTOCOL_TLSv1_2

Selects TLS version 1.2 as the channel encryption protocol. This is the most modern version, and probably the best choice for maximum protection, if both sides can speak it. Available only with openssl version 1.0.1+.

Nouveau dans la version 3.4.

Obsolète depuis la version 3.6 : OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.OP_ALL

Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL's *SSL_OP_ALL* constant.

Nouveau dans la version 3.2.

ssl.OP_NO_SSLv2

Prevents an SSLv2 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing SSLv2 as the protocol version.

Nouveau dans la version 3.2.

Obsolète depuis la version 3.6 : SSLv2 is deprecated

`ssl.OP_NO_SSLv3`

Prevents an SSLv3 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing SSLv3 as the protocol version.

Nouveau dans la version 3.2.

Obsolète depuis la version 3.6 : SSLv3 is deprecated

`ssl.OP_NO_TLSv1`

Prevents a TLSv1 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1 as the protocol version.

Nouveau dans la version 3.2.

`ssl.OP_NO_TLSv1_1`

Prevents a TLSv1.1 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

Nouveau dans la version 3.4.

`ssl.OP_NO_TLSv1_2`

Prevents a TLSv1.2 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

Nouveau dans la version 3.4.

`ssl.OP_NO_TLSv1_3`

Prevents a TLSv1.3 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to 0.

Nouveau dans la version 3.6.3.

`ssl.OP_CIPHER_SERVER_PREFERENCE`

Use the server's cipher ordering preference, rather than the client's. This option has no effect on client sockets and SSLv2 server sockets.

Nouveau dans la version 3.3.

`ssl.OP_SINGLE_DH_USE`

Prevents re-use of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Nouveau dans la version 3.3.

`ssl.OP_SINGLE_ECDH_USE`

Prevents re-use of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Nouveau dans la version 3.3.

`ssl.OP_ENABLE_MIDDLEBOX_COMPAT`

Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

This option is only available with OpenSSL 1.1.1 and later.

Nouveau dans la version 3.6.7.

`ssl.OP_NO_COMPRESSION`

Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

This option is only available with OpenSSL 1.0.0 and later.

Nouveau dans la version 3.3.

`class ssl.Options`

enum.IntFlag collection of `OP_*` constants.

ssl.OP_NO_TICKET

Prevent client side from requesting a session ticket.
Nouveau dans la version 3.6.

ssl.HAS_ALPN

Whether the OpenSSL library has built-in support for the *Application-Layer Protocol Negotiation* TLS extension as described in [RFC 7301](#).
Nouveau dans la version 3.5.

ssl.HAS_ECDH

Whether the OpenSSL library has built-in support for Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.
Nouveau dans la version 3.3.

ssl.HAS_SNI

Whether the OpenSSL library has built-in support for the *Server Name Indication* extension (as defined in [RFC 6066](#)).
Nouveau dans la version 3.2.

ssl.HAS_NPN

Whether the OpenSSL library has built-in support for *Next Protocol Negotiation* as described in the [NPN draft specification](#). When true, you can use the `SSLContext.set_npn_protocols()` method to advertise which protocols you want to support.
Nouveau dans la version 3.3.

ssl.HAS_TLSv1_3

Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.
Nouveau dans la version 3.6.3.

ssl.CHANNEL_BINDING_TYPES

List of supported TLS channel binding types. Strings in this list can be used as arguments to `SSLSocket.get_channel_binding()`.
Nouveau dans la version 3.3.

ssl.OPENSSSL_VERSION

The version string of the OpenSSL library loaded by the interpreter :

```
>>> ssl.OPENSSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

Nouveau dans la version 3.2.

ssl.OPENSSSL_VERSION_INFO

A tuple of five integers representing version information about the OpenSSL library :

```
>>> ssl.OPENSSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

Nouveau dans la version 3.2.

ssl.OPENSSSL_VERSION_NUMBER

The raw version number of the OpenSSL library, as a single integer :

```
>>> ssl.OPENSSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSSL_VERSION_NUMBER)
'0x100020bf'
```

Nouveau dans la version 3.2.

ssl.ALERT_DESCRIPTION_HANDSHAKE_FAILURE

`ssl.ALERT_DESCRIPTION_INTERNAL_ERROR`

ALERT_DESCRIPTION_*

Alert Descriptions from [RFC 5246](#) and others. The [IANA TLS Alert Registry](#) contains this list and references to the RFCs where their meaning is defined.

Used as the return value of the callback function in `SSLContext.set_servername_callback()`.

Nouveau dans la version 3.4.

class `ssl.AlertDescription`

`enum.IntEnum` collection of `ALERT_DESCRIPTION_*` constants.

Nouveau dans la version 3.6.

Purpose. **SERVER_AUTH**

Option for `create_default_context()` and `SSLContext.load_default_certs()`. This value indicates that the context may be used to authenticate Web servers (therefore, it will be used to create client-side sockets).

Nouveau dans la version 3.4.

Purpose. **CLIENT_AUTH**

Option for `create_default_context()` and `SSLContext.load_default_certs()`. This value indicates that the context may be used to authenticate Web clients (therefore, it will be used to create server-side sockets).

Nouveau dans la version 3.4.

class `ssl.SSLErrorNumber`

`enum.IntEnum` collection of `SSL_ERROR_*` constants.

Nouveau dans la version 3.6.

18.2.2 SSL Sockets

class `ssl.SSLSocket` (*socket.socket*)

SSL sockets provide the following methods of *Socket Objects* :

- `accept()`
- `bind()`
- `close()`
- `connect()`
- `detach()`
- `fileno()`
- `getpeername()`, `getsockname()`
- `getsockopt()`, `setsockopt()`
- `gettimeout()`, `settimeout()`, `setblocking()`
- `listen()`
- `makefile()`
- `recv()`, `recv_into()` (but passing a non-zero `flags` argument is not allowed)
- `send()`, `sendall()` (with the same limitation)
- `sendfile()` (but `os.sendfile` will be used for plain-text sockets only, else `send()` will be used)
- `shutdown()`

However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the *notes on non-blocking sockets*.

Usually, `SSLSocket` are not created directly, but using the `SSLContext.wrap_socket()` method.

Modifié dans la version 3.5 : The `sendfile()` method was added.

Modifié dans la version 3.5 : The `shutdown()` does not reset the socket timeout each time bytes are received or sent. The socket timeout is now to maximum total duration of the shutdown.

Obsolète depuis la version 3.6 : It is deprecated to create a `SSLSocket` instance directly, use `SSLContext.wrap_socket()` to wrap a socket.

SSL sockets also have the following additional methods and attributes :

`SSLSocket.read(len=1024, buffer=None)`

Read up to `len` bytes of data from the SSL socket and return the result as a `bytes` instance. If `buffer` is specified, then read into the buffer instead, and return the number of bytes read.

Raise `SSLWantReadError` or `SSLWantWriteError` if the socket is *non-blocking* and the read would block. As at any time a re-negotiation is possible, a call to `read()` can also cause write operations.

Modifié dans la version 3.5 : The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration to read up to `len` bytes.

Obsolète depuis la version 3.6 : Use `recv()` instead of `read()`.

`SSLSocket.write(buf)`

Write `buf` to the SSL socket and return the number of bytes written. The `buf` argument must be an object supporting the buffer interface.

Raise `SSLWantReadError` or `SSLWantWriteError` if the socket is *non-blocking* and the write would block. As at any time a re-negotiation is possible, a call to `write()` can also cause read operations.

Modifié dans la version 3.5 : The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration to write `buf`.

Obsolète depuis la version 3.6 : Use `send()` instead of `write()`.

Note : The `read()` and `write()` methods are the low-level methods that read and write unencrypted, application-level data and decrypt/encrypt it to encrypted, wire-level data. These methods require an active SSL connection, i.e. the handshake was completed and `SSLSocket.unwrap()` was not called.

Normally you should use the socket API methods like `recv()` and `send()` instead of these methods.

`SSLSocket.do_handshake()`

Perform the SSL setup handshake.

Modifié dans la version 3.4 : The handshake method also performs `match_hostname()` when the `check_hostname` attribute of the socket's `context` is true.

Modifié dans la version 3.5 : The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration of the handshake.

`SSLSocket.getpeercert(binary_form=False)`

If there is no certificate for the peer on the other end of the connection, return `None`. If the SSL handshake hasn't been done yet, raise `ValueError`.

If the `binary_form` parameter is `False`, and a certificate was received from the peer, this method returns a `dict` instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them `subject` (the principal for which the certificate was issued) and `issuer` (the principal issuing the certificate). If a certificate contains an instance of the *Subject Alternative Name* extension (see [RFC 3280](#)), there will also be a `subjectAltName` key in the dictionary.

The `subject` and `issuer` fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate's data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example :

```
{'issuer': (((('countryName', 'IL'),),
              (('organizationName', 'StartCom Ltd.'),),
              (('organizationalUnitName',
               'Secure Digital Certificate Signing'),),
              (('commonName',
               'StartCom Class 2 Primary Intermediate Server CA'),)),
```

(suite sur la page suivante)

(suite de la page précédente)

```
'notAfter': 'Nov 22 08:15:19 2013 GMT',
'notBefore': 'Nov 21 03:09:52 2011 GMT',
'serialNumber': '95F0',
'subject': (((('description', '571208-SLe257oHY9fVQ07Z'),),
              (('countryName', 'US'),),
              (('stateOrProvinceName', 'California'),),
              (('localityName', 'San Francisco'),),
              (('organizationName', 'Electronic Frontier Foundation, Inc. '),),
              (('commonName', '*.eff.org'),),
              (('emailAddress', 'hostmaster@eff.org'),)),
'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
'version': 3}
```

Note : To validate a certificate for a particular service, you can use the `match_hostname()` function.

If the `binary_form` parameter is `True`, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or `None` if the peer did not provide a certificate. Whether the peer provides a certificate depends on the SSL socket's role :

- for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required ;
- for a server SSL socket, the client will only provide a certificate when requested by the server ; therefore `getpeercert()` will return `None` if you used `CERT_NONE` (rather than `CERT_OPTIONAL` or `CERT_REQUIRED`).

Modifié dans la version 3.2 : The returned dictionary includes additional items such as `issuer` and `notBefore`.

Modifié dans la version 3.4 : `ValueError` is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and OCSP URIs.

`SSLSocket.cipher()`

Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`.

`SSLSocket.shared_ciphers()`

Return the list of ciphers shared by the client during the handshake. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. `shared_ciphers()` returns `None` if no connection has been established or the socket is a client socket.

Nouveau dans la version 3.5.

`SSLSocket.compression()`

Return the compression algorithm being used as a string, or `None` if the connection isn't compressed.

If the higher-level protocol supports its own compression mechanism, you can use `OP_NO_COMPRESSION` to disable SSL-level compression.

Nouveau dans la version 3.3.

`SSLSocket.get_channel_binding(cb_type="tls-unique")`

Get channel binding data for current connection, as a bytes object. Returns `None` if not connected or the handshake has not been completed.

The `cb_type` parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the `CHANNEL_BINDING_TYPES` list. Currently only the "tls-unique" channel binding, defined by **RFC 5929**, is supported. `ValueError` will be raised if an unsupported channel binding type is requested.

Nouveau dans la version 3.3.

`SSLSocket.selected_alpn_protocol()`

Return the protocol that was selected during the TLS handshake. If `SSLContext.set_alpn_protocols()` was not called, if the other party does not support ALPN, if this socket does not support any of the client's proposed protocols, or if the handshake has not happened yet, `None` is returned.

Nouveau dans la version 3.5.

`SSLSocket.selected_npn_protocol()`

Return the higher-level protocol that was selected during the TLS/SSL handshake. If `SSLContext.set_npn_protocols()` was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

Nouveau dans la version 3.3.

`SSLSocket.unwrap()`

Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

`SSLSocket.verify_client_post_handshake()`

Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3 connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see `SSLContext.post_handshake_auth`.

The method does not perform a cert exchange immediately. The server-side sends a `CertificateRequest` during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn't met (e.g. not TLS 1.3, PHA not enabled), an `SSL_ERROR` is raised.

Nouveau dans la version 3.6.7.

Note : Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises `NotImplementedError`.

`SSLSocket.version()`

Return the actual SSL protocol version negotiated by the connection as a string, or `None` if no secure connection is established. As of this writing, possible return values include `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`. Recent OpenSSL versions may define more return values.

Nouveau dans la version 3.5.

`SSLSocket.pending()`

Returns the number of already decrypted bytes available for read, pending on the connection.

`SSLSocket.context`

The `SSLContext` object this SSL socket is tied to. If the SSL socket was created using the top-level `wrap_socket()` function (rather than `SSLContext.wrap_socket()`), this is a custom context object created for this SSL socket.

Nouveau dans la version 3.2.

`SSLSocket.server_side`

A boolean which is `True` for server-side sockets and `False` for client-side sockets.

Nouveau dans la version 3.2.

`SSLSocket.server_hostname`

Hostname of the server : `str` type, or `None` for server-side socket or if the hostname was not specified in the constructor.

Nouveau dans la version 3.2.

`SSLSocket.session`

The `SSLSession` for this SSL connection. The session is available for client and server side sockets after the TLS handshake has been performed. For client sockets the session can be set before `do_handshake()` has been called to reuse a session.

Nouveau dans la version 3.6.

`SSLSocket.session_reused`

Nouveau dans la version 3.6.

18.2.3 SSL Contexts

Nouveau dans la version 3.2.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

class `ssl.SSLContext` (*protocol=PROTOCOL_TLS*)

Create a new SSL context. You may pass *protocol* which must be one of the `PROTOCOL_*` constants defined in this module. `PROTOCOL_TLS` is currently recommended for maximum interoperability and default value.

Voir aussi :

`create_default_context()` lets the `ssl` module choose security settings for a given purpose.

Modifié dans la version 3.6 : The context is created with secure default values. The options `OP_NO_COMPRESSION`, `OP_CIPHER_SERVER_PREFERENCE`, `OP_SINGLE_DH_USE`, `OP_SINGLE_ECDH_USE`, `OP_NO_SSLv2` (except for `PROTOCOL_SSLv2`), and `OP_NO_SSLv3` (except for `PROTOCOL_SSLv3`) are set by default. The initial cipher suite list contains only HIGH ciphers, no NULL ciphers and no MD5 ciphers (except for `PROTOCOL_SSLv2`).

`SSLContext` objects have the following methods and attributes :

`SSLContext.cert_store_stats()`

Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert :

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

Nouveau dans la version 3.4.

`SSLContext.load_cert_chain` (*certfile, keyfile=None, password=None*)

Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The *keyfile* string, if present, must point to a file containing the private key in. Otherwise the private key will be taken from *certfile* as well. See the discussion of [Certificates](#) for more information on how the certificate is stored in the *certfile*.

The *password* argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the *password* argument. It will be ignored if the private key is not encrypted and no password is needed.

If the *password* argument is not specified and a password is required, OpenSSL's built-in password prompting mechanism will be used to interactively prompt the user for a password.

An `SSLError` is raised if the private key doesn't match with the certificate.

Modifié dans la version 3.3 : New optional argument *password*.

`SSLContext.load_default_certs` (*purpose=Purpose.SERVER_AUTH*)

Load a set of default « certification authority » (CA) certificates from default locations. On Windows it loads CA certs from the CA and ROOT system stores. On other systems it calls `SSLContext.set_default_verify_paths()`. In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings `Purpose.SERVER_AUTH` loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). `Purpose.CLIENT_AUTH` loads CA certificates for client certificate verification on the server side.

Nouveau dans la version 3.4.

`SSLContext.load_verify_locations` (*cafile=None, capath=None, cadata=None*)

Load a set of « certification authority » (CA) certificates used to validate other peers' certificates when *verify_mode* is other than `CERT_NONE`. At least one of *cafile* or *capath* must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, `SSLContext.verify_flags` must be configured properly.

The *cafile* string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of *Certificates* for more information about how to arrange the certificates in this file.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an *OpenSSL specific layout*.

The *cadata* object, if present, is either an ASCII string of one or more PEM-encoded certificates or a *bytes-like object* of DER-encoded certificates. Like with *capath* extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

Modifié dans la version 3.4 : New optional argument *cadata*

`SSLContext.get_ca_certs` (*binary_form=False*)

Get a list of loaded « certification authority » (CA) certificates. If the *binary_form* parameter is *False* each list entry is a dict like the output of `SSLSocket.getpeercert()`. Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from *capath* unless a certificate was requested and loaded by a SSL connection.

Note : Certificates in a *capath* directory aren't loaded unless they have been used at least once.

Nouveau dans la version 3.4.

`SSLContext.get_ciphers` ()

Get a list of enabled ciphers. The list is in order of cipher priority. See `SSLContext.set_ciphers()`.

Exemple :

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers() # OpenSSL 1.0.x
[{'alg_bits': 256,
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH      Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'id': 50380848,
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 256},
 {'alg_bits': 128,
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH      Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'id': 50380847,
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 128}]
```

On OpenSSL 1.1 and newer the cipher dict contains additional fields ::

```
>>> ctx.get_ciphers() # OpenSSL 1.1+
[{'aead': True,
```

(suite sur la page suivante)

(suite de la page précédente)

```

'alg_bits': 256,
'auth': 'auth-rsa',
'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH      Au=RSA  '
                'Enc=AESGCM(256) Mac=AEAD',
'digest': None,
'id': 50380848,
'kea': 'kx-ecdhe',
'name': 'ECDHE-RSA-AES256-GCM-SHA384',
'protocol': 'TLSv1.2',
'strength_bits': 256,
'symmetric': 'aes-256-gcm'},
{'aead': True,
 'alg_bits': 128,
 'auth': 'auth-rsa',
 'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH      Au=RSA  '
                'Enc=AESGCM(128) Mac=AEAD',
 'digest': None,
 'id': 50380847,
 'kea': 'kx-ecdhe',
 'name': 'ECDHE-RSA-AES128-GCM-SHA256',
 'protocol': 'TLSv1.2',
 'strength_bits': 128,
 'symmetric': 'aes-128-gcm'}]]

```

Availability : OpenSSL 1.0.2+

Nouveau dans la version 3.6.

`SSLContext.set_default_verify_paths()`

Load a set of default « certification authority » (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there's no easy way to know whether this method succeeds : no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

`SSLContext.set_ciphers(ciphers)`

Set the available ciphers for sockets created with this context. It should be a string in the [OpenSSL cipher list format](#). If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an `SSL_ERROR` will be raised.

Note : when connected, the `SSLSocket.cipher()` method of SSL sockets will give the currently selected cipher.

OpenSSL 1.1.1 has TLS 1.3 cipher suites enabled by default. The suites cannot be disabled with `set_ciphers()`.

`SSLContext.set_alpn_protocols(protocols)`

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to [RFC 7301](#). After a successful handshake, the `SSLSocket.selected_alpn_protocol()` method will return the agreed-upon protocol.

This method will raise `NotImplementedError` if `HAS_ALPN` is False.

OpenSSL 1.1.0 to 1.1.0e will abort the handshake and raise `SSL_ERROR` when both sides support ALPN but cannot agree on a protocol. 1.1.0f+ behaves like 1.0.2, `SSLSocket.selected_alpn_protocol()` returns None.

Nouveau dans la version 3.5.

`SSLContext.set_npn_protocols(protocols)`

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like

`['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the [NPN draft specification](#). After a successful handshake, the `SSLSocket.selected_npn_protocol()` method will return the agreed-upon protocol.

This method will raise `NotImplementedError` if `HAS_NPN` is `False`.

Nouveau dans la version 3.3.

`SSLContext.set_servername_callback(server_name_callback)`

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in [RFC 6066](#) section 3 - Server Name Indication.

Only one callback can be set per `SSLContext`. If `server_name_callback` is `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function, `server_name_callback`, will be called with three arguments; the first being the `ssl.SSLSocket`, the second is a string that represents the server name that the client is intending to communicate (or `None` if the TLS Client Hello does not contain a server name) and the third argument is the original `SSLContext`. The server name argument is the IDNA decoded server name.

A typical use of this callback is to change the `ssl.SSLSocket`'s `SSLSocket.context` attribute to a new object of type `SSLContext` representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like `SSLSocket.selected_alpn_protocol()` and `SSLSocket.context`. `SSLSocket.getpeercert()`, `SSLSocket.getpeername()`, `SSLSocket.cipher()` and `SSLSocket.compress()` methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not contain return meaningful values nor can they be called safely.

The `server_name_callback` function must return `None` to allow the TLS negotiation to continue. If a TLS failure is required, a constant `ALERT_DESCRIPTION_*` can be returned. Other return values will result in a TLS fatal error with `ALERT_DESCRIPTION_INTERNAL_ERROR`.

If there is an IDNA decoding error on the server name, the TLS connection will terminate with an `ALERT_DESCRIPTION_INTERNAL_ERROR` fatal TLS alert message to the client.

If an exception is raised from the `server_name_callback` function the TLS connection will terminate with a fatal TLS alert message `ALERT_DESCRIPTION_HANDSHAKE_FAILURE`.

This method will raise `NotImplementedError` if the OpenSSL library had `OPENSSL_NO_TLSEXT` defined when it was built.

Nouveau dans la version 3.4.

`SSLContext.load_dh_params(dhfile)`

Load the key generation parameters for Diffie-Hellman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The `dhfile` parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn't apply to client sockets. You can also use the `OP_SINGLE_DH_USE` option to further improve security.

Nouveau dans la version 3.3.

`SSLContext.set_ecdh_curve(curve_name)`

Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The `curve_name` parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn't apply to client sockets. You can also use the `OP_SINGLE_ECDH_USE` option to further improve security.

This method is not available if `HAS_ECDH` is `False`.

Nouveau dans la version 3.3.

Voir aussi :

[SSL/TLS & Perfect Forward Secrecy](#) Vincent Bernat.

`SSLContext.wrap_socket(sock, server_side=False, do_handshake_on_connect=True, suppress_ragged_eofs=True, server_hostname=None, session=None)`

Wrap an existing Python socket *sock* and return an *SSLSocket* object. *sock* must be a *SOCK_STREAM* socket; other socket types are unsupported.

The returned SSL socket is tied to the context, its settings and certificates. The parameters *server_side*, *do_handshake_on_connect* and *suppress_ragged_eofs* have the same meaning as in the top-level *wrap_socket()* function.

On client connections, the optional parameter *server_hostname* specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying *server_hostname* will raise a *ValueError* if *server_side* is true.

session, see *session*.

Modifié dans la version 3.5 : Always allow a *server_hostname* to be passed, even if OpenSSL does not have SNI.

Modifié dans la version 3.6 : *session* argument was added.

`SSLContext.wrap_bio(incoming, outgoing, server_side=False, server_hostname=None, session=None)`

Create a new *SSLObject* instance by wrapping the BIO objects *incoming* and *outgoing*. The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

The *server_side*, *server_hostname* and *session* parameters have the same meaning as in *SSLContext.wrap_socket()*.

Modifié dans la version 3.6 : *session* argument was added.

`SSLContext.session_stats()`

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each *piece of information* to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created :

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

`SSLContext.check_hostname`

Whether to match the peer cert's hostname with *match_hostname()* in *SSLSocket.do_handshake()*. The context's *verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, and you must pass *server_hostname* to *wrap_socket()* in order to match the hostname.

Exemple :

```
import socket, ssl

context = ssl.SSLContext()
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

Nouveau dans la version 3.4.

Note : This features requires OpenSSL 0.9.8f or newer.

`SSLContext.options`

An integer representing the set of SSL options enabled on this context. The default value is *OP_ALL*, but you can specify other options such as *OP_NO_SSLv2* by ORing them together.

Note : With versions of OpenSSL older than 0.9.8m, it is only possible to set options, not to clear them. Attempting to clear an option (by resetting the corresponding bits) will raise a `ValueError`.

Modifié dans la version 3.6 : `SSLContext.options` returns `Options` flags :

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

`SSLContext.post_handshake_auth`

Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, `SSLContext.verify_mode` must be set to `CERT_OPTIONAL` or `CERT_REQUIRED`, too. The actual client cert exchange is delayed until `SSLSocket.verify_client_post_handshake()` is called and some I/O is performed.

Nouveau dans la version 3.6.7.

Note : Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the property value is `None` and can't be modified

`SSLContext.protocol`

The protocol version chosen when constructing the context. This attribute is read-only.

`SSLContext.verify_flags`

The flags for certificate verification operations. You can set flags like `VERIFY_CRL_CHECK_LEAF` by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs). Available only with openssl version 0.9.8+.

Nouveau dans la version 3.4.

Modifié dans la version 3.6 : `SSLContext.verify_flags` returns `VerifyFlags` flags :

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

`SSLContext.verify_mode`

Whether to try to verify other peers' certificates and how to behave if verification fails. This attribute must be one of `CERT_NONE`, `CERT_OPTIONAL` or `CERT_REQUIRED`.

Modifié dans la version 3.6 : `SSLContext.verify_mode` returns `VerifyMode` enum :

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

18.2.4 Certificates

Certificates in general are part of a public-key / private-key system. In this system, each *principal*, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the *public key*; the other part is kept secret, and is called the *private key*. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and **only** with the other part.

A certificate contains information about two principals. It contains the name of a *subject*, and the subject's public key. It also contains a statement by a second principal, the *issuer*, that the subject is who they claim to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer knows.

However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called « notBefore » and « notAfter ».

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework ; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as « PEM » (see [RFC 1422](#)), which is a base-64 encoded form wrapped with a header line and a footer line :

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a *certificate chain*. This chain should start with the specific certificate for the principal who « is » the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of *that* certificate, and so on up the chain till you get to a certificate which is *self-signed*, that is, a certificate which has the same subject and issuer, sometimes called a *root certificate*. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority's certificate :

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA certificates

If you are going to require validation of the other side of the connection's certificate, you need to provide a « CA certs » file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform's certificates file can be used by calling `SSLContext.load_default_certs()`, this is done automatically with `create_default_context()`.

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to `SSLContext.load_cert_chain()` and `wrap_socket()` needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain :

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following :

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.....++++++
.....++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

18.2.5 Examples

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom :

```
try:
    import ssl
except ImportError:
    pass
else:
    ... # do something that requires SSL support
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification :

```
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right) :

```
>>> context = ssl.SSLContext()
>>> context.verify_mode = ssl.CERT_REQUIRED
>>> context.check_hostname = True
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in `/etc/ssl/certs/ca-bundle.crt`; if not, you'll get an error and have to adjust the location)

When you use the context to connect to a server, `CERT_REQUIRED` validates the server certificate : it ensures that the server certificate was signed with one of the CA certificates, and checks the signature for correctness :

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                             server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

You may then fetch the certificate :

```
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`):

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': (('countryName', 'US'),,
            ('organizationName', 'DigiCert Inc'),,
            ('organizationalUnitName', 'www.digicert.com'),,
            ('commonName', 'DigiCert SHA2 Extended Validation Server CA'))),
```

(suite sur la page suivante)

(suite de la page précédente)

```
'notAfter': 'Sep  9 12:00:00 2016 GMT',
'notBefore': 'Sep  5 00:00:00 2014 GMT',
'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
'subject': (((('businessCategory', 'Private Organization'),),
              (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
              (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
              (('serialNumber', '3359300'),),
              (('streetAddress', '16 Allen Rd'),),
              (('postalCode', '03894-4801'),),
              (('countryName', 'US'),),
              (('stateOrProvinceName', 'NH'),),
              (('localityName', 'Wolfeboro'),),
              (('organizationName', 'Python Software Foundation'),),
              (('commonName', 'www.python.org'),)),
'subjectAltName': (('DNS', 'www.python.org'),
                  ('DNS', 'python.org'),
                  ('DNS', 'pypi.org'),
                  ('DNS', 'docs.python.org'),
                  ('DNS', 'testpypi.org'),
                  ('DNS', 'bugs.python.org'),
                  ('DNS', 'wiki.python.org'),
                  ('DNS', 'hg.python.org'),
                  ('DNS', 'mail.python.org'),
                  ('DNS', 'packaging.python.org'),
                  ('DNS', 'pythonhosted.org'),
                  ('DNS', 'www.pythonhosted.org'),
                  ('DNS', 'test.pythonhosted.org'),
                  ('DNS', 'us.pycon.org'),
                  ('DNS', 'id.python.org')),
'version': 3}
```

Now the SSL channel is established and the certificate verified, you can proceed to talk with the server :

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

See the discussion of *Security considerations* below.

Server-side operation

For server operation, typically you'll need to have a server certificate, and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call `listen()` on it, and start waiting for clients to connect :

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

When a client connects, you'll call `accept()` on the socket to get the new socket from the other end, and use the context's `SSLContext.wrap_socket()` method to create a server-side SSL socket for the connection :

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you) :

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in *non-blocking mode* and use an event loop).

18.2.6 Notes on non-blocking sockets

SSL sockets behave slightly different than regular sockets in non-blocking mode. When working with non-blocking sockets, there are thus several things you need to be aware of :

- Most `SSLSocket` methods will raise either `SSLWantWriteError` or `SSLWantReadError` instead of `BlockingIOError` if an I/O operation would block. `SSLWantReadError` will be raised if a read operation on the underlying socket is necessary, and `SSLWantWriteError` for a write operation on the underlying socket. Note that attempts to *write* to an SSL socket may require *reading* from the underlying socket first, and attempts to *read* from the SSL socket may require a prior *write* to the underlying socket.
Modifié dans la version 3.5 : In earlier Python versions, the `SSLSocket.send()` method returned zero instead of raising `SSLWantWriteError` or `SSLWantReadError`.

- Calling `select()` tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to `select()`.
- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without `select()` being aware of it. Therefore, you should first call `SSLSocket.recv()` to drain any potentially available data, and then only block on a `select()` call if still necessary.
(of course, similar provisions apply when using other primitives such as `poll()`, or those in the `selectors` module)
- The SSL handshake itself will be non-blocking : the `SSLSocket.do_handshake()` method has to be retried until it returns successfully. Here is a synopsis using `select()` to wait for the socket's readiness :

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

Voir aussi :

The `asyncio` module supports *non-blocking SSL sockets* and provides a higher level API. It polls for events using the `selectors` module and handles `SSLWantWriteError`, `SSLWantReadError` and `BlockingIOError` exceptions. It runs the SSL handshake asynchronously as well.

18.2.7 Memory BIO Support

Nouveau dans la version 3.5.

Ever since the SSL module was introduced in Python 2.6, the `SSLSocket` class has provided two related but distinct areas of functionality :

- SSL protocol handling
- Network IO

The network IO API is identical to that provided by `socket.socket`, from which `SSLSocket` also inherits. This allows an SSL socket to be used as a drop-in replacement for a regular socket, making it very easy to add SSL support to an existing application.

Combining SSL protocol handling and network IO usually works well, but there are some cases where it doesn't. An example is async IO frameworks that want to use a different IO multiplexing model than the « select/poll on a file descriptor » (readiness based) model that is assumed by `socket.socket` and by the internal OpenSSL socket IO routines. This is mostly relevant for platforms like Windows where this model is not efficient. For this purpose, a reduced scope variant of `SSLSocket` called `SSLObject` is provided.

class `ssl.SSLObject`

A reduced-scope variant of `SSLSocket` representing an SSL protocol instance that does not contain any network IO methods. This class is typically used by framework authors that want to implement asynchronous IO for SSL through memory buffers.

This class implements an interface on top of a low-level SSL object as implemented by OpenSSL. This object captures the state of an SSL connection but does not provide any network IO itself. IO needs to be performed through separate « BIO » objects which are OpenSSL's IO abstraction layer.

An `SSLObject` instance can be created using the `wrap_bio()` method. This method will create the `SSLObject` instance and bind it to a pair of BIOs. The *incoming* BIO is used to pass data from Python to the SSL protocol instance, while the *outgoing* BIO is used to pass data the other way around.

The following methods are available :

- `context`
- `server_side`
- `server_hostname`
- `session`
- `session_reused`
- `read()`
- `write()`
- `getpeercert()`
- `selected_npn_protocol()`
- `cipher()`
- `shared_ciphers()`
- `compression()`
- `pending()`
- `do_handshake()`
- `unwrap()`
- `get_channel_binding()`

When compared to `SSLSocket`, this object lacks the following features :

- Any form of network IO; `recv()` and `send()` read and write only to the underlying `MemoryBIO` buffers.
- There is no `do_handshake_on_connect` machinery. You must always manually call `do_handshake()` to start the handshake.
- There is no handling of `suppress_ragged_eofs`. All end-of-file conditions that are in violation of the protocol are reported via the `SSLEOFError` exception.
- The method `unwrap()` call does not return anything, unlike for an SSL socket where it returns the underlying socket.
- The `server_name_callback` callback passed to `SSLContext.set_servername_callback()` will get an `SSLObject` instance instead of a `SSLSocket` instance as its first parameter.

Some notes related to the use of `SSLObject` :

- All IO on an `SSLObject` is *non-blocking*. This means that for example `read()` will raise an `SSLWantReadError` if it needs more data than the incoming BIO has available.
- There is no module-level `wrap_bio()` call like there is for `wrap_socket()`. An `SSLObject` is always created via an `SSLContext`.

An `SSLObject` communicates with the outside world using memory buffers. The class `MemoryBIO` provides a memory buffer that can be used for this purpose. It wraps an OpenSSL memory BIO (Basic IO) object :

class `ssl.MemoryBIO`

A memory buffer that can be used to pass data between Python and an SSL protocol instance.

pending

Return the number of bytes currently in the memory buffer.

eof

A boolean indicating whether the memory BIO is current at the end-of-file position.

read (*n=-1*)

Read up to *n* bytes from the memory buffer. If *n* is not specified or negative, all bytes are returned.

write (*buf*)

Write the bytes from *buf* to the memory BIO. The *buf* argument must be an object supporting the buffer protocol.

The return value is the number of bytes written, which is always equal to the length of *buf*.

write_eof ()

Write an EOF marker to the memory BIO. After this method has been called, it is illegal to call `write()`. The attribute `eof` will become true after all data currently in the buffer has been read.

18.2.8 SSL session

Nouveau dans la version 3.6.

```
class ssl.SSLSession
    Session object used by session.
    id
    time
    timeout
    ticket_lifetime_hint
    has_ticket
```

18.2.9 Security considerations

Best defaults

For **client use**, if you don't have any special requirements for your security policy, it is highly recommended that you use the `create_default_context()` function to create your SSL context. It will load the system's trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

For example, here is how you would use the `smtpplib.SMTP` class to create a trusted, secure connection to a SMTP server :

```
>>> import ssl, smtpplib
>>> smtp = smtpplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

If a client certificate is needed for the connection, it can be added with `SSLContext.load_cert_chain()`.

By contrast, if you create the SSL context by calling the `SSLContext` constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

Manual settings

Verifying certificates

When calling the `SSLContext` constructor directly, `CERT_NONE` is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use `CERT_REQUIRED`. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling `SSLSocket.getpeercert()`, matches the desired service. For many protocols and applications, the service can be identified by the hostname; in this case, the `match_hostname()` function can be used. This common check is automatically performed when `SSLContext.check_hostname` is enabled.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify `CERT_REQUIRED` and similarly check the client certificate.

Protocol versions

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER` as the protocol version. SSLv2 and SSLv3 are disabled by default.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.options |= ssl.OP_NO_TLSv1
>>> client_context.options |= ssl.OP_NO_TLSv1_1
```

The SSL context created above will only allow TLSv1.2 and later (if supported by your system) connections to a server. `PROTOCOL_TLS_CLIENT` implies certificate validation and hostname checks by default. You have to load certificates into the context.

Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the `SSLContext.set_ciphers()` method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL's documentation about the [cipher list format](#). If you want to check which ciphers are enabled by a given cipher list, use `SSLContext.get_ciphers()` or the `openssl ciphers` command on your system.

Multi-processing

If using this module as part of a multi-processed application (using, for example the `multiprocessing` or `concurrent.futures` modules), be aware that OpenSSL's internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with `os.fork()`. Any successful call of `RAND_add()`, `RAND_bytes()` or `RAND_pseudo_bytes()` is sufficient.

18.2.10 LibreSSL support

LibreSSL is a fork of OpenSSL 1.0.1. The ssl module has limited support for LibreSSL. Some features are not available when the ssl module is compiled with LibreSSL.

- LibreSSL >= 2.6.1 no longer supports NPN. The methods `SSLContext.set_npn_protocols()` and `SSLSocket.selected_npn_protocol()` are not available.
- `SSLContext.set_default_verify_paths()` ignores the env vars `SSL_CERT_FILE` and `SSL_CERT_PATH` although `get_default_verify_paths()` still reports them.

Voir aussi :

Class `socket.socket` Documentation of underlying `socket` class

SSL/TLS Strong Encryption : An Introduction Intro from the Apache HTTP Server documentation

RFC 1422 : Privacy Enhancement for Internet Electronic Mail : Part II : Certificate-Based Key Management
Steve Kent

RFC 4086 : Randomness Requirements for Security Donald E., Jeffrey I. Schiller

RFC 5280 : Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
D. Cooper

RFC 5246 : The Transport Layer Security (TLS) Protocol Version 1.2 T. Dierks et. al.

RFC 6066 : Transport Layer Security (TLS) Extensions D. Eastlake

IANA TLS : Transport Layer Security (TLS) Parameters IANA

RFC 7525 : Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)
IETF

Mozilla's Server Side TLS recommendations Mozilla

18.3 `select` — Waiting for I/O completion

This module provides access to the `select()` and `poll()` functions available in most operating systems, `devpoll()` available on Solaris and derivatives, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

Note : The `selectors` module allows high-level and efficient I/O multiplexing, built upon the `select` module primitives. Users are encouraged to use the `selectors` module instead, unless they want precise control over the OS-level primitives used.

Le module définit :

exception `select.error`

A deprecated alias of `OSError`.

Modifié dans la version 3.3 : Following **PEP 3151**, this class was made an alias of `OSError`.

`select.devpoll()`

(Only supported on Solaris and derivatives.) Returns a `/dev/poll` polling object; see section [/dev/poll Polling Objects](#) below for the methods supported by `devpoll` objects.

`devpoll()` objects are linked to the number of file descriptors allowed at the time of instantiation. If your program reduces this value, `devpoll()` will fail. If your program increases this value, `devpoll()` may return an incomplete list of active file descriptors.

The new file descriptor is *non-inheritable*.

Nouveau dans la version 3.3.

Modifié dans la version 3.4 : Le nouveau descripteur de fichier est maintenant non-héritable.

`select.epoll(sizehint=-1, flags=0)`

(Only supported on Linux 2.5.44 and newer.) Return an edge polling object, which can be used as Edge or Level Triggered interface for I/O events.

`sizehint` informs `epoll` about the expected number of events to be registered. It must be positive, or `-1` to use the default. It is only used on older systems where `epoll_create1()` is not available; otherwise it has no effect (though its value is still checked).

`flags` is deprecated and completely ignored. However, when supplied, its value must be `0` or `select.EPOLL_CLOEXEC`, otherwise `OSError` is raised.

See the [Edge and Level Trigger Polling \(epoll\) Objects](#) section below for the methods supported by epolling objects.

`epoll` objects support the context management protocol : when used in a `with` statement, the new file descriptor is automatically closed at the end of the block.

The new file descriptor is *non-inheritable*.

Modifié dans la version 3.3 : Added the `flags` parameter.

Modifié dans la version 3.4 : Support for the `with` statement was added. The new file descriptor is now non-inheritable.

Obsolète depuis la version 3.4 : The `flags` parameter. `select.EPOLL_CLOEXEC` is used by default now. Use `os.set_inheritable()` to make the file descriptor inheritable.

`select.poll()`

(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section [Polling Objects](#) below for the methods supported by polling objects.

`select.kqueue()`

(Only supported on BSD.) Returns a kernel queue object; see section [Kqueue Objects](#) below for the methods supported by kqueue objects.

The new file descriptor is *non-inheritable*.

Modifié dans la version 3.4 : Le nouveau descripteur de fichier est maintenant non-héritable.

`select.kevent(ident, filter=KQ_FILTER_READ, flags=KQ_EV_ADD, fflags=0, data=0, udata=0)`

(Only supported on BSD.) Returns a kernel event object; see section [Kevent Objects](#) below for the methods supported by kevent objects.

`select.select(rlist, wlist, xlist[, timeout])`

This is a straightforward interface to the Unix `select()` system call. The first three arguments are sequences of “waitable objects”: either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer:

— `rlist`: wait until ready for reading

— `wlist`: wait until ready for writing

— `xlist`: wait for an « exceptional condition » (see the manual page for what your system considers such a condition)

Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on Unix but not on Windows.) The optional `timeout` argument specifies a time-out as a floating point number in seconds. When the `timeout` argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the sequences are Python *file objects* (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a *wrapper* class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer).

Note : File objects on Windows are not acceptable, but sockets are. On Windows, the underlying `select()` function is provided by the WinSock library, and does not handle file descriptors that don’t originate from WinSock.

Modifié dans la version 3.5 : The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see [PEP 475](#) for the rationale), instead of raising `InterruptedError`.

`select.PIPE_BUF`

The minimum number of bytes which can be written without blocking to a pipe when the pipe has been reported as ready for writing by `select()`, `poll()` or another interface in this module. This doesn’t apply to other kind of file-like objects such as sockets.

This value is guaranteed by POSIX to be at least 512. Availability : Unix.

Nouveau dans la version 3.2.

18.3.1 /dev/poll Polling Objects

Solaris and derivatives have `/dev/poll`. While `select()` is $O(\text{highest file descriptor})$ and `poll()` is $O(\text{number of file descriptors})$, `/dev/poll` is $O(\text{active file descriptors})$.

`/dev/poll` behaviour is very close to the standard `poll()` object.

`devpoll.close()`

Close the file descriptor of the polling object.

Nouveau dans la version 3.4.

`devpoll.closed`

True if the polling object is closed.

Nouveau dans la version 3.4.

`devpoll.fileno()`

Return the file descriptor number of the polling object.

Nouveau dans la version 3.4.

`devpoll.register(fd[, eventmask])`

Register a file descriptor with the polling object. Future calls to the `poll()` method will then check whether the file descriptor has any pending I/O events. `fd` can be either an integer, or an object with a `fileno()` method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

`eventmask` is an optional bitmask describing the type of events you want to check for. The constants are the same that with `poll()` object. The default value is a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`.

Avertissement : Registering a file descriptor that's already registered is not an error, but the result is undefined. The appropriate action is to unregister or modify it first. This is an important difference compared with `poll()`.

`devpoll.modify(fd[, eventmask])`

This method does an `unregister()` followed by a `register()`. It is (a bit) more efficient than doing the same explicitly.

`devpoll.unregister(fd)`

Remove a file descriptor being tracked by a polling object. Just like the `register()` method, `fd` can be an integer or an object with a `fileno()` method that returns an integer.

Attempting to remove a file descriptor that was never registered is safely ignored.

`devpoll.poll([timeout])`

Polls the set of registered file descriptors, and returns a possibly-empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. `fd` is the file descriptor, and `event` is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If `timeout` is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If `timeout` is omitted, -1, or `None`, the call will block until there is an event for this poll object.

Modifié dans la version 3.5 : The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see [PEP 475](#) for the rationale), instead of raising `InterruptedError`.

18.3.2 Edge and Level Trigger Polling (epoll) Objects

<http://linux.die.net/man/4/epoll>

eventmask

Constante	Signification
EPOLLIN	Available for read
EPOLLOUT	Available for write
EPOLLPRI	Urgent data for read
EPOLLERR	Error condition happened on the assoc. fd
EPOLLHUP	Hang up happened on the assoc. fd
EPOLLET	Set Edge Trigger behavior, the default is Level Trigger behavior
EPOLLONESHOT	Set one-shot behavior. After one event is pulled out, the fd is internally disabled
EPOLLEXCLUSIVE	Wake only one epoll object when the associated fd has an event. The default (if this flag is not set) is to wake all epoll objects polling on a fd.
EPOLLRDHUP	Stream socket peer closed connection or shut down writing half of connection.
EPOLLRDNORMAL	Equivalent to EPOLLIN
EPOLLRDBAND	Priority data band can be read.
EPOLLWRNORMAL	Equivalent to EPOLLOUT
EPOLLWRBAND	Priority data may be written.
EPOLMSG	Ignored.

`epoll.close()`

Close the control file descriptor of the epoll object.

`epoll.closed`

True if the epoll object is closed.

`epoll.fileno()`

Return the file descriptor number of the control fd.

`epoll.fromfd(fd)`

Create an epoll object from a given file descriptor.

`epoll.register(fd[, eventmask])`

Register a fd descriptor with the epoll object.

`epoll.modify(fd, eventmask)`

Modify a registered file descriptor.

`epoll.unregister(fd)`

Remove a registered file descriptor from the epoll object.

`epoll.poll(timeout=-1, maxevents=-1)`

Wait for events. timeout in seconds (float)

Modifié dans la version 3.5 : The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see [PEP 475](#) for the rationale), instead of raising *InterruptedError*.

18.3.3 Polling Objects

The `poll()` system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. `poll()` scales better because the system call only requires listing the file descriptors of interest, while `select()` builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. `select()` is $O(\text{highest file descriptor})$, while `poll()` is $O(\text{number of file descriptors})$.

`poll.register(fd[, eventmask])`

Register a file descriptor with the polling object. Future calls to the `poll()` method will then check whether the file descriptor has any pending I/O events. `fd` can be either an integer, or an object with a `fileno()` method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

`eventmask` is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`, described in the table below. If not specified, the default value used will check for all 3 types of events.

Constante	Signification
<code>POLLIN</code>	There is data to read
<code>POLLPRI</code>	There is urgent data to read
<code>POLLOUT</code>	Ready for output : writing will not block
<code>POLLERR</code>	Error condition of some sort
<code>POLLHUP</code>	Hung up
<code>POLLRDHUP</code>	Stream socket peer closed connection, or shut down writing half of connection
<code>POLLNVAL</code>	Invalid request : descriptor not open

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

`poll.modify(fd, eventmask)`

Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an `OSError` exception with `errno ENOENT` to be raised.

`poll.unregister(fd)`

Remove a file descriptor being tracked by a polling object. Just like the `register()` method, `fd` can be an integer or an object with a `fileno()` method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a `KeyError` exception to be raised.

`poll.poll([timeout])`

Polls the set of registered file descriptors, and returns a possibly-empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. `fd` is the file descriptor, and `event` is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If `timeout` is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If `timeout` is omitted, negative, or `None`, the call will block until there is an event for this poll object.

Modifié dans la version 3.5 : The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see [PEP 475](#) for the rationale), instead of raising `InterruptedError`.

18.3.4 Kqueue Objects

`kqueue.close()`

Close the control file descriptor of the kqueue object.

`kqueue.closed`

True if the kqueue object is closed.

`kqueue.fileno()`

Return the file descriptor number of the control fd.

`kqueue.fromfd(fd)`

Create a kqueue object from a given file descriptor.

`kqueue.control(changelist, max_events[, timeout=None])` → eventlist

Low level interface to kevent

— changelist must be an iterable of kevent object or None

— max_events must be 0 or a positive integer

— timeout in seconds (floats possible)

Modifié dans la version 3.5 : The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see [PEP 475](#) for the rationale), instead of raising *InterruptedError*.

18.3.5 Kevent Objects

<https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2>

`kevent.ident`

Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor ident can either be an int or an object with a *fileno()* method. kevent stores the integer internally.

`kevent.filter`

Name of the kernel filter.

Constante	Signification
<code>KQ_FILTER_READ</code>	Takes a descriptor and returns whenever there is data available to read
<code>KQ_FILTER_WRITE</code>	Takes a descriptor and returns whenever there is data available to write
<code>KQ_FILTER_AIO</code>	AIO requests
<code>KQ_FILTER_VNODE</code>	Returns when one or more of the requested events watched in <i>fflag</i> occurs
<code>KQ_FILTER_PROC</code>	Watch for events on a process id
<code>KQ_FILTER_NETDEV</code>	Watch for events on a network device [not available on Mac OS X]
<code>KQ_FILTER_SIGNAL</code>	Returns whenever the watched signal is delivered to the process
<code>KQ_FILTER_TIMER</code>	Establishes an arbitrary timer

`kevent.flags`

Filter action.

Constante	Signification
KQ_EV_ADD	Adds or modifies an event
KQ_EV_DELETE	Removes an event from the queue
KQ_EV_ENABLE	Permits <code>control()</code> to return the event
KQ_EV_DISABLE	Disables event
KQ_EV_ONESHOT	Removes event after first occurrence
KQ_EV_CLEAR	Reset the state after an event is retrieved
KQ_EV_SYSFLAGS	internal event
KQ_EV_FLAG1	internal event
KQ_EV_EOF	Filter specific EOF condition
KQ_EV_ERROR	See return values

`kevent.fflags`

Filter specific flags.

KQ_FILTER_READ and KQ_FILTER_WRITE filter flags :

Constante	Signification
KQ_NOTE_LOWAT	low water mark of a socket buffer

KQ_FILTER_VNODE filter flags :

Constante	Signification
KQ_NOTE_DELETE	<i>unlink()</i> was called
KQ_NOTE_WRITE	a write occurred
KQ_NOTE_EXTEND	the file was extended
KQ_NOTE_ATTRIB	an attribute was changed
KQ_NOTE_LINK	the link count has changed
KQ_NOTE_RENAME	the file was renamed
KQ_NOTE_REVOKE	access to the file was revoked

KQ_FILTER_PROC filter flags :

Constante	Signification
KQ_NOTE_EXIT	the process has exited
KQ_NOTE_FORK	the process has called <i>fork()</i>
KQ_NOTE_EXEC	the process has executed a new process
KQ_NOTE_PCTRLMASK	internal filter flag
KQ_NOTE_PDATAMASK	internal filter flag
KQ_NOTE_TRACK	follow a process across <i>fork()</i>
KQ_NOTE_CHILD	returned on the child process for <i>NOTE_TRACK</i>
KQ_NOTE_TRACKERR	unable to attach to a child

KQ_FILTER_NETDEV filter flags (not available on Mac OS X) :

Constante	Signification
KQ_NOTE_LINKUP	link is up
KQ_NOTE_LINKDOWN	link is down
KQ_NOTE_LINKINV	link state is invalid

`kevent.data`

Filter specific data.

`kevent.udata`

User defined value.

18.4 selectors — High-level I/O multiplexing

Nouveau dans la version 3.4.

Source code : [Lib/selectors.py](#)

18.4.1 Introduction

This module allows high-level and efficient I/O multiplexing, built upon the `select` module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.

It defines a `BaseSelector` abstract base class, along with several concrete implementations (`KqueueSelector`, `EpollSelector`...), that can be used to wait for I/O readiness notification on multiple file objects. In the following, « file object » refers to any object with a `fileno()` method, or a raw file descriptor. See [file object](#).

`DefaultSelector` is an alias to the most efficient implementation available on the current platform : this should be the default choice for most users.

Note : The type of file objects supported depends on the platform : on Windows, sockets are supported, but not pipes, whereas on Unix, both are supported (some other types may be supported as well, such as fifos or special file devices).

Voir aussi :

`select` Low-level I/O multiplexing module.

18.4.2 Classes

Classes hierarchy :

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

In the following, `events` is a bitwise mask indicating which I/O events should be waited for on a given file object. It can be a combination of the modules constants below :

Constante	Signification
<code>EVENT_READ</code>	Available for read
<code>EVENT_WRITE</code>	Available for write

class `selectors.SelectorKey`

A `SelectorKey` is a [namedtuple](#) used to associate a file object to its underlying file descriptor, selected event mask and attached data. It is returned by several `BaseSelector` methods.

fileobj

File object registered.

fd

Underlying file descriptor.

events

Events that must be waited for on this file object.

data

Optional opaque data associated to this file object : for example, this could be used to store a per-client session ID.

class selectors.**BaseSelector**

A *BaseSelector* is used to wait for I/O event readiness on multiple file objects. It supports file stream registration, unregistration, and a method to wait for I/O events on those streams, with an optional timeout. It's an abstract base class, so cannot be instantiated. Use *DefaultSelector* instead, or one of *SelectSelector*, *KqueueSelector* etc. if you want to specifically use an implementation, and your platform supports it. *BaseSelector* and its concrete implementations support the *context manager* protocol.

abstractmethod **register** (*fileobj*, *events*, *data=None*)

Register a file object for selection, monitoring it for I/O events.

fileobj is the file object to monitor. It may either be an integer file descriptor or an object with a `fileno()` method. *events* is a bitwise mask of events to monitor. *data* is an opaque object.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is already registered.

abstractmethod **unregister** (*fileobj*)

Unregister a file object from selection, removing it from monitoring. A file object shall be unregistered prior to being closed.

fileobj must be a file object previously registered.

This returns the associated *SelectorKey* instance, or raises a *KeyError* if *fileobj* is not registered. It will raise *ValueError* if *fileobj* is invalid (e.g. it has no `fileno()` method or its `fileno()` method has an invalid return value).

modify (*fileobj*, *events*, *data=None*)

Change a registered file object's monitored events or attached data.

This is equivalent to `BaseSelector.unregister(fileobj)()` followed by `BaseSelector.register(fileobj, events, data)()`, except that it can be implemented more efficiently.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is not registered.

abstractmethod **select** (*timeout=None*)

Wait until some registered file objects become ready, or the timeout expires.

If *timeout* > 0, this specifies the maximum wait time, in seconds. If *timeout* <= 0, the call won't block, and will report the currently ready file objects. If *timeout* is `None`, the call will block until a monitored file object becomes ready.

This returns a list of (*key*, *events*) tuples, one for each ready file object.

key is the *SelectorKey* instance corresponding to a ready file object. *events* is a bitmask of events ready on this file object.

Note : This method can return before any file object becomes ready or the timeout has elapsed if the current process receives a signal : in this case, an empty list will be returned.

Modifié dans la version 3.5 : The selector is now retried with a recomputed timeout when interrupted by a signal if the signal handler did not raise an exception (see [PEP 475](#) for the rationale), instead of returning an empty list of events before the timeout.

close ()

Close the selector.

This must be called to make sure that any underlying resource is freed. The selector shall not be used once it has been closed.

get_key (*fileobj*)

Return the key associated with a registered file object.

This returns the `SelectorKey` instance associated to this file object, or raises `KeyError` if the file object is not registered.

abstractmethod `get_map()`

Return a mapping of file objects to selector keys.

This returns a `Mapping` instance mapping registered file objects to their associated `SelectorKey` instance.

class `selectors.DefaultSelector`

The default selector class, using the most efficient implementation available on the current platform. This should be the default choice for most users.

class `selectors.SelectSelector`

`select.select()`-based selector.

class `selectors.PollSelector`

`select.poll()`-based selector.

class `selectors.EpollSelector`

`select.epoll()`-based selector.

fileno()

This returns the file descriptor used by the underlying `select.epoll()` object.

class `selectors.DevpollSelector`

`select.devpoll()`-based selector.

fileno()

This returns the file descriptor used by the underlying `select.devpoll()` object.

Nouveau dans la version 3.5.

class `selectors.KqueueSelector`

`select.kqueue()`-based selector.

fileno()

This returns the file descriptor used by the underlying `select.kqueue()` object.

18.4.3 Exemples

Here is a simple echo server implementation :

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept() # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000) # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data) # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
```

(suite sur la page suivante)

(suite de la page précédente)

```

conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)

```

18.5 asyncio — Entrées/Sorties asynchrone, boucle d'évènements, coroutines et tâches

Nouveau dans la version 3.4.

Code source : [Lib/asyncio/](#)

Ce module fournit l'infrastructure pour écrire des programmes à fil d'exécution unique (*single-thread en anglais*) mais permettant l'exécution de code concurrent en utilisant les coroutines, les accès multiplexés aux entrées-sorties par l'intermédiaire de *sockets* ou autres ressources, la gestion de clients et serveurs réseaux et d'autres fonctions primitives associées. Voici une liste plus détaillée du contenu du paquet :

- une *boucle d'évènements* prête à l'emploi dont les implémentations sont spécifiques à leur plateforme ;
- Des abstractions pour les couches *transport* et *protocole* (similaire à celles proposées par *Twisted*) ;
- pour la gestion effective de TCP, UDP, SSL, la communication inter-processus par tubes, les appels différés, et autres (certains peuvent être dépendant du système) ;
- une classe *Future* qui imite celle du *concurrent.futures* module, mais qui est adaptée pour fonctionner avec la boucle d'évènements ;
- des coroutines et tâches qui se basent sur `yield from` (**PEP 380**), pour écrire du code concurrent de manière séquentielle ;
- annulation de la gestion de la classe *Futures* et coroutines ;
- *des primitives de synchronisation* à utiliser entre des coroutines dans un fil d'exécution unique, en imitant celles présentes dans le module *threading* ;
- une interface pour déléguer des tâches à un groupe de fils d'exécutions, lorsque vous avez absolument besoin d'utiliser une bibliothèque qui effectue des entrées-sorties bloquantes.

Programmer de façon asynchrone est plus complexe que programmer d'une façon séquentielle : lisez la page [Develop with asyncio](#) qui liste les pièges fréquents et explique la manière de les éviter. *Activer le mode de débogage d'asyncio* pendant le développement afin de détecter les problèmes courants.

Table des matières :

18.5.1 Base Event Loop

Source code : [Lib/asyncio/events.py](#)

The event loop is the central execution device provided by *asyncio*. It provides multiple facilities, including :

- Registering, executing and cancelling delayed calls (timeouts).
- Creating client and server *transports* for various kinds of communication.
- Launching subprocesses and the associated *transports* for communication with an external program.
- Delegating costly function calls to a pool of threads.

class `asyncio.BaseEventLoop`

This class is an implementation detail. It is a subclass of *AbstractEventLoop* and may be a base class of concrete event loop implementations found in *asyncio*. It should not be used directly; use *AbstractEventLoop* instead. *BaseEventLoop* should not be subclassed by third-party code; the internal interface is not stable.

class `asyncio.AbstractEventLoop`

Abstract base class of event loops.

This class is *not thread safe*.

Exécuter une boucle d'évènements

`AbstractEventLoop.run_forever()`

Run until *stop()* is called. If *stop()* is called before *run_forever()* is called, this polls the I/O selector once with a timeout of zero, runs all callbacks scheduled in response to I/O events (and those that were already scheduled), and then exits. If *stop()* is called while *run_forever()* is running, this will run the current batch of callbacks and then exit. Note that callbacks scheduled by callbacks will not run in that case; they will run the next time *run_forever()* is called.

Modifié dans la version 3.5.1.

`AbstractEventLoop.run_until_complete(future)`

Exécuter jusqu'à ce que *Future* soit terminé.

If the argument is a *coroutine object*, it is wrapped by *ensure_future()*.

Return the Future's result, or raise its exception.

`AbstractEventLoop.is_running()`

Donne le status d'exécution de la boucle d'évènements.

`AbstractEventLoop.stop()`

Arrête l'exécution de la boucle d'évènements.

This causes *run_forever()* to exit at the next suitable opportunity (see there for more details).

Modifié dans la version 3.5.1.

`AbstractEventLoop.is_closed()`

Donne True si la boucle d'évènements est fermée.

Nouveau dans la version 3.4.2.

`AbstractEventLoop.close()`

Close the event loop. The loop must not be running. Pending callbacks will be lost.

This clears the queues and shuts down the executor, but does not wait for the executor to finish.

This is idempotent and irreversible. No other methods should be called after this one.

coroutine `AbstractEventLoop.shutdown_asyncgens()`

Schedule all currently open *asynchronous generator* objects to close with an *aclose()* call. After calling this method, the event loop will issue a warning whenever a new asynchronous generator is iterated. Should be used to finalize all scheduled asynchronous generators reliably. Example :


```

try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()

```

Nouveau dans la version 3.6.

Appels

Most *asyncio* functions don't accept keywords. If you want to pass keywords to your callback, use *functools.partial()*. For example, `loop.call_soon(functools.partial(print, "Hello", flush=True))` will call `print("Hello", flush=True)`.

Note : *functools.partial()* is better than lambda functions, because *asyncio* can inspect *functools.partial()* object to display parameters in debug mode, whereas lambda functions have a poor representation.

`AbstractEventLoop.call_soon(callback, *args)`

Arrange for a callback to be called as soon as possible. The callback is called after *call_soon()* returns, when control returns to the event loop.

This operates as a FIFO queue, callbacks are called in the order in which they are registered. Each callback will be called exactly once.

Any positional arguments after the callback will be passed to the callback when it is called.

An instance of *asyncio.Handle* is returned, which can be used to cancel the callback.

Use `functools.partial` to pass keywords to the callback.

`AbstractEventLoop.call_soon_threadsafe(callback, *args)`

Comme *call_soon()* mais *thread safe*.

Voir la section *exécution concurrente et multi-fils d'exécution* de la documentation.

Appels différés

The event loop has its own internal clock for computing timeouts. Which clock is used depends on the (platform-specific) event loop implementation; ideally it is a monotonic clock. This will generally be a different clock than *time.time()*.

Note : Timeouts (relative *delay* or absolute *when*) should not exceed one day.

`AbstractEventLoop.call_later(delay, callback, *args)`

Arrange for the *callback* to be called after the given *delay* seconds (either an int or float).

An instance of *asyncio.Handle* is returned, which can be used to cancel the callback.

callback will be called exactly once per call to *call_later()*. If two callbacks are scheduled for exactly the same time, it is undefined which will be called first.

The optional positional *args* will be passed to the callback when it is called. If you want the callback to be called with some named arguments, use a closure or *functools.partial()*.

Use `functools.partial` to pass keywords to the callback.

`AbstractEventLoop.call_at(when, callback, *args)`

Arrange for the *callback* to be called at the given absolute timestamp *when* (an int or float), using the same time reference as *AbstractEventLoop.time()*.

This method's behavior is the same as *call_later()*.

An instance of `asyncio.Handle` is returned, which can be used to cancel the callback.

Use `functools.partial` to pass keywords to the callback.

`AbstractEventLoop.time()`

Return the current time, as a *float* value, according to the event loop's internal clock.

Voir aussi :

La fonction `asyncio.sleep()`.

Futurs

`AbstractEventLoop.create_future()`

Create an *asyncio.Future* object attached to the loop.

This is a preferred way to create futures in asyncio, as event loop implementations can provide alternative implementations of the Future class (with better performance or instrumentation).

Nouveau dans la version 3.5.2.

Tâches

`AbstractEventLoop.create_task(coro)`

Schedule the execution of a *coroutine object* : wrap it in a future. Return a *Task* object.

Third-party event loops can use their own subclass of *Task* for interoperability. In this case, the result type is a subclass of *Task*.

This method was added in Python 3.4.2. Use the `async()` function to support also older Python versions.

Nouveau dans la version 3.4.2.

`AbstractEventLoop.set_task_factory(factory)`

Set a task factory that will be used by `AbstractEventLoop.create_task()`.

If *factory* is *None* the default task factory will be set.

If *factory* is a *callable*, it should have a signature matching `(loop, coro)`, where *loop* will be a reference to the active event loop, *coro* will be a coroutine object. The callable must return an *asyncio.Future* compatible object.

Nouveau dans la version 3.4.4.

`AbstractEventLoop.get_task_factory()`

Return a task factory, or *None* if the default one is in use.

Nouveau dans la version 3.4.4.

Créer des connections

coroutine `AbstractEventLoop.create_connection(protocol_factory, host=None, port=None, *, ssl=None, family=0, proto=0, flags=0, sock=None, local_addr=None, server_hostname=None)`

Create a streaming transport connection to a given Internet *host* and *port* : socket family *AF_INET* or *AF_INET6* depending on *host* (or *family* if specified), socket type *SOCK_STREAM*. *protocol_factory* must be a callable returning a *protocol* instance.

This method is a *coroutine* which will try to establish the connection in the background. When successful, the coroutine returns a `(transport, protocol)` pair.

The chronological synopsis of the underlying operation is as follows :

1. The connection is established, and a *transport* is created to represent it.

2. *protocol_factory* is called without arguments and must return a *protocol* instance.
3. The protocol instance is tied to the transport, and its `connection_made()` method is called.
4. The coroutine returns successfully with the `(transport, protocol)` pair.

The created transport is an implementation-dependent bidirectional stream.

Note : *protocol_factory* can be any kind of callable, not necessarily a class. For example, if you want to use a pre-created protocol instance, you can pass `lambda: my_protocol`.

Options modifiant la création de la connexion :

- *ssl* : if given and not false, a SSL/TLS transport is created (by default a plain TCP transport is created). If *ssl* is a *ssl.SSLContext* object, this context is used to create the transport; if *ssl* is *True*, a context with some unspecified default settings is used.

Voir aussi :

SSL/TLS security considerations

- *server_hostname*, is only for use together with *ssl*, and sets or overrides the hostname that the target server's certificate will be matched against. By default the value of the *host* argument is used. If *host* is empty, there is no default and you must pass a value for *server_hostname*. If *server_hostname* is an empty string, hostname matching is disabled (which is a serious security risk, allowing for man-in-the-middle-attacks).
- *family*, *proto*, *flags* are the optional address family, protocol and flags to be passed through to `getaddrinfo()` for *host* resolution. If given, these should all be integers from the corresponding *socket* module constants.
- *sock*, if given, should be an existing, already connected *socket.socket* object to be used by the transport. If *sock* is given, none of *host*, *port*, *family*, *proto*, *flags* and *local_addr* should be specified.
- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket to locally. The *local_host* and *local_port* are looked up using `getaddrinfo()`, similarly to *host* and *port*.

Modifié dans la version 3.5 : On Windows with *ProactorEventLoop*, SSL/TLS is now supported.

Voir aussi :

The *open_connection()* function can be used to get a pair of `(StreamReader, StreamWriter)` instead of a protocol.

```
coroutine AbstractEventLoop.create_datagram_endpoint (protocol_factory,          lo-
                                                    cal_addr=None,          re-
                                                    mote_addr=None,          *, fa-
                                                    mily=0, proto=0, flags=0,
                                                    reuse_address=None,
                                                    reuse_port=None,          al-
                                                    low_broadcast=None,
                                                    sock=None)
```

Note : The parameter *reuse_address* is no longer supported, as using `SO_REUSEADDR` poses a significant security concern for UDP. Explicitly passing *reuse_address=True* will raise an exception.

When multiple processes with differing UIDs assign sockets to an identical UDP socket address with `SO_REUSEADDR`, incoming packets can become randomly distributed among the sockets.

For supported platforms, *reuse_port* can be used as a replacement for similar functionality. With *reuse_port*, `SO_REUSEPORT` is used instead, which specifically prevents processes with differing UIDs from assigning sockets to the same socket address.

Create a datagram connection.

Create datagram connection : socket family *AF_INET* or *AF_INET6* depending on *host* (or *family* if specified), socket type *SOCK_DGRAM*. *protocol_factory* must be a callable returning a *protocol* instance.

This method is a *coroutine* which will try to establish the connection in the background. When successful, the coroutine returns a `(transport, protocol)` pair.

Options modifiant la création de la connexion :

- *local_addr*, if given, is a (*local_host*, *local_port*) tuple used to bind the socket to locally. The *local_host* and *local_port* are looked up using `getaddrinfo()`.
- *remote_addr*, if given, is a (*remote_host*, *remote_port*) tuple used to connect the socket to a remote address. The *remote_host* and *remote_port* are looked up using `getaddrinfo()`.
- *family*, *proto*, *flags* are the optional address family, protocol and flags to be passed through to `getaddrinfo()` for *host* resolution. If given, these should all be integers from the corresponding *socket* module constants.
- *reuse_port* tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows and some UNIX's. If the `SO_REUSEPORT` constant is not defined then this capability is unsupported.
- *allow_broadcast* tells the kernel to allow this endpoint to send messages to the broadcast address.
- *sock* can optionally be specified in order to use a preexisting, already connected, *socket.socket* object to be used by the transport. If specified, *local_addr* and *remote_addr* should be omitted (must be *None*).

On Windows with *ProactorEventLoop*, this method is not supported.

See *UDP echo client protocol* and *UDP echo server protocol* examples.

Modifié dans la version 3.4.4 : The *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, **allow_broadcast*, and *sock* parameters were added.

Modifié dans la version 3.6.10 : The *reuse_address* parameter is no longer supporter due to security concerns

[illegible]

Create UNIX connection : socket family `AF_UNIX`, socket type `SOCK_STREAM`. The `AF_UNIX` socket family is used to communicate between processes on the same machine efficiently.

This method is a *coroutine* which will try to establish the connection in the background. When successful, the coroutine returns a `(transport, protocol)` pair.

path is the name of a UNIX domain socket, and is required unless a *sock* parameter is specified. Abstract UNIX sockets, *str*, and *bytes* paths are supported.

See the `AbstractEventLoop.create_connection()` method for parameters.

Disponible sur : UNIX.

Attendre des connections

```
coroutine AbstractEventLoop.create_server (protocol_factory,      host=None,      port=None,
*,
                                          family=socket.AF_UNSPEC,
flags=socket.AI_PASSIVE, sock=None, backlog=100,
ssl=None, reuse_address=None,
reuse_port=None)
```

Create a TCP server (socket type `SOCK_STREAM`) bound to *host* and *port*.

Return a `Server` object, its `sockets` attribute contains created sockets. Use the `Server.close()` method to stop the server : close listening sockets.

Paramètres :

- The *host* parameter can be a string, in that case the TCP server is bound to *host* and *port*. The *host* parameter can also be a sequence of strings and in that case the TCP server is bound to all hosts of the sequence. If *host* is an empty string or `None`, all interfaces are assumed and a list of multiple sockets will be returned (most likely one for IPv4 and another one for IPv6).
- *family* can be set to either `socket.AF_INET` or `AF_INET6` to force the socket to use IPv4 or IPv6. If not set it will be determined from host (defaults to `socket.AF_UNSPEC`).
- *flags* est un masque de bits pour `getaddrinfo()`.
- *sock* can optionally be specified in order to use a preexisting socket object. If specified, *host* and *port* should be omitted (must be `None`).
- *backlog* is the maximum number of queued connections passed to `listen()` (defaults to 100).
- *ssl* can be set to an `SSLContext` to enable SSL over the accepted connections.

- *reuse_address* tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire. If not specified will automatically be set to `True` on UNIX.
- *reuse_port* tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows.

Cette méthode est une *coroutine*.

Modifié dans la version 3.5 : On Windows with *ProactorEventLoop*, SSL/TLS is now supported.

Voir aussi :

The function *start_server()* creates a (*StreamReader*, *StreamWriter*) pair and calls back a function with this pair.

Modifié dans la version 3.5.1 : The *host* parameter can now be a sequence of strings.

coroutine *AbstractEventLoop.create_unix_server* (*protocol_factory*, *path=None*, *, *sock=None*, *backlog=100*, *ssl=None*)

Similar to *AbstractEventLoop.create_server()*, but specific to the socket family *AF_UNIX*.

Cette méthode est une *coroutine*.

Disponible sur : UNIX.

coroutine *BaseEventLoop.connect_accepted_socket* (*protocol_factory*, *sock*, *, *ssl=None*)

Handle an accepted connection.

This is used by servers that accept connections outside of *asyncio* but that use *asyncio* to handle them.

Paramètres :

- *sock* is a preexisting socket object returned from an *accept* call.
- *ssl* can be set to an *SSLContext* to enable SSL over the accepted connections.

This method is a *coroutine*. When completed, the coroutine returns a (*transport*, *protocol*) pair.

Nouveau dans la version 3.5.3.

Surveiller des descripteurs de fichiers

On Windows with *SelectorEventLoop*, only socket handles are supported (ex : pipe file descriptors are not supported).

On Windows with *ProactorEventLoop*, these methods are not supported.

AbstractEventLoop.add_reader (*fd*, *callback*, **args*)

Start watching the file descriptor for read availability and then call the *callback* with specified arguments.

Use functools.partial to pass keywords to the callback.

AbstractEventLoop.remove_reader (*fd*)

Stop watching the file descriptor for read availability.

AbstractEventLoop.add_writer (*fd*, *callback*, **args*)

Start watching the file descriptor for write availability and then call the *callback* with specified arguments.

Use functools.partial to pass keywords to the callback.

AbstractEventLoop.remove_writer (*fd*)

Stop watching the file descriptor for write availability.

The *watch a file descriptor for read events* example uses the low-level *AbstractEventLoop.add_reader()* method to register the file descriptor of a socket.

Opérations bas niveau sur les *socket*

coroutine `AbstractEventLoop.sock_recv(sock, nbytes)`

Receive data from the socket. Modeled after blocking `socket.socket.recv()` method.

The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *nbytes*.

With `SelectorEventLoop` event loop, the socket *sock* must be non-blocking.

Cette méthode est une *coroutine*.

coroutine `AbstractEventLoop.sock_sendall(sock, data)`

Send data to the socket. Modeled after blocking `socket.socket.sendall()` method.

The socket must be connected to a remote socket. This method continues to send data from *data* until either all data has been sent or an error occurs. `None` is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully processed by the receiving end of the connection.

With `SelectorEventLoop` event loop, the socket *sock* must be non-blocking.

Cette méthode est une *coroutine*.

coroutine `AbstractEventLoop.sock_connect(sock, address)`

Connect to a remote socket at *address*. Modeled after blocking `socket.socket.connect()` method.

With `SelectorEventLoop` event loop, the socket *sock* must be non-blocking.

Cette méthode est une *coroutine*.

Modifié dans la version 3.5.2 : *address* no longer needs to be resolved. `sock_connect` will try to check if the *address* is already resolved by calling `socket.inet_pton()`. If not, `AbstractEventLoop.getaddrinfo()` will be used to resolve the *address*.

Voir aussi :

`AbstractEventLoop.create_connection()` and `asyncio.open_connection()`.

coroutine `AbstractEventLoop.sock_accept(sock)`

Accept a connection. Modeled after blocking `socket.socket.accept()`.

The socket must be bound to an address and listening for connections. The return value is a pair (*conn*, *address*) where *conn* is a new socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

La *socket sock* ne doit pas être bloquante.

Cette méthode est une *coroutine*.

Voir aussi :

`AbstractEventLoop.create_server()` and `start_server()`.

Résout le nom d'hôte

coroutine `AbstractEventLoop.getaddrinfo(host, port, *, family=0, type=0, proto=0, flags=0)`

This method is a *coroutine*, similar to `socket.getaddrinfo()` function but non-blocking.

coroutine `AbstractEventLoop.getnameinfo(sockaddr, flags=0)`

This method is a *coroutine*, similar to `socket.getnameinfo()` function but non-blocking.

Connect pipes

On Windows with *SelectorEventLoop*, these methods are not supported. Use *ProactorEventLoop* to support pipes on Windows.

coroutine *AbstractEventLoop.connect_read_pipe* (*protocol_factory*, *pipe*)

Register read pipe in eventloop.

protocol_factory should instantiate object with *Protocol* interface. *pipe* is a *file-like object*. Return pair (*transport*, *protocol*), where *transport* supports the *ReadTransport* interface.

With *SelectorEventLoop* event loop, the *pipe* is set to non-blocking mode.

Cette méthode est une *coroutine*.

coroutine *AbstractEventLoop.connect_write_pipe* (*protocol_factory*, *pipe*)

Register write pipe in eventloop.

protocol_factory should instantiate object with *BaseProtocol* interface. *pipe* is *file-like object*. Return pair (*transport*, *protocol*), where *transport* supports *WriteTransport* interface.

With *SelectorEventLoop* event loop, the *pipe* is set to non-blocking mode.

Cette méthode est une *coroutine*.

Voir aussi :

The *AbstractEventLoop.subprocess_exec()* and *AbstractEventLoop.subprocess_shell()* methods.

Signaux UNIX

Disponibilité : UNIX seulement.

AbstractEventLoop.add_signal_handler (*signum*, *callback*, **args*)

Ajouter un gestionnaire (*handler*) pour un signal.

Raise *ValueError* if the signal number is invalid or uncatchable. Raise *RuntimeError* if there is a problem setting up the handler.

Use functools.partial to pass keywords to the callback.

AbstractEventLoop.remove_signal_handler (*sig*)

Supprimer un *handler* pour un signal.

Return True if a signal handler was removed, False if not.

Voir aussi :

Le module *signal*.

Exécuteur

Call a function in an *Executor* (pool of threads or pool of processes). By default, an event loop uses a thread pool executor (*ThreadPoolExecutor*).

coroutine *AbstractEventLoop.run_in_executor* (*executor*, *func*, **args*)

Arrange for a *func* to be called in the specified executor.

The *executor* argument should be an *Executor* instance. The default executor is used if *executor* is None.

*Use functools.partial to pass keywords to the *func*.*

Cette méthode est une *coroutine*.

Modifié dans la version 3.5.3 : *BaseEventLoop.run_in_executor()* no longer configures the *max_workers* of the thread pool executor it creates, instead leaving it up to the thread pool executor (*ThreadPoolExecutor*) to set the default.

`AbstractEventLoop.set_default_executor(executor)`

Set the default executor used by `run_in_executor()`.

API de gestion d'erreur

Allows customizing how exceptions are handled in the event loop.

`AbstractEventLoop.set_exception_handler(handler)`

Set *handler* as the new event loop exception handler.

If *handler* is `None`, the default exception handler will be set.

If *handler* is a callable object, it should have a matching signature to `(loop, context)`, where `loop` will be a reference to the active event loop, `context` will be a dict object (see `call_exception_handler()` documentation for details about context).

`AbstractEventLoop.get_exception_handler()`

Return the exception handler, or `None` if the default one is in use.

Nouveau dans la version 3.5.2.

`AbstractEventLoop.default_exception_handler(context)`

Gestionnaire d'exception par défaut.

This is called when an exception occurs and no exception handler is set, and can be called by a custom exception handler that wants to defer to the default behavior.

context parameter has the same meaning as in `call_exception_handler()`.

`AbstractEventLoop.call_exception_handler(context)`

Appelle le gestionnaire d'exception de la boucle d'évènements actuelle.

context is a dict object containing the following keys (new keys may be introduced later) :

- `message` : Message d'erreur ;
- `exception` (optionnel) : Un objet exception ;
- `"future"` (optional) : `asyncio.Future` instance ;
- `"handle"` (optional) : `asyncio.Handle` instance ;
- `"protocol"` (optional) : `Protocol` instance ;
- `"transport"` (optional) : `Transport` instance ;
- `"socket"` (optional) : `socket.socket` instance.

Note : Note : this method should not be overloaded in subclassed event loops. For any custom exception handling, use `set_exception_handler()` method.

Mode débogage

`AbstractEventLoop.get_debug()`

Get the debug mode (*bool*) of the event loop.

The default value is `True` if the environment variable `PYTHONASYNCIODEBUG` is set to a non-empty string, `False` otherwise.

Nouveau dans la version 3.4.2.

`AbstractEventLoop.set_debug(enabled : bool)`

Active le mode débogage pour la boucle d'évènements.

Nouveau dans la version 3.4.2.

Voir aussi :

The *debug mode of asyncio*.

Serveur

class `asyncio.Server`

Serveur écoutant sur des *sockets*.

Object created by the `AbstractEventLoop.create_server()` method and the `start_server()` function. Don't instantiate the class directly.

close()

Stop serving : close listening sockets and set the `sockets` attribute to `None`.

The sockets that represent existing incoming client connections are left open.

The server is closed asynchronously, use the `wait_closed()` coroutine to wait until the server is closed.

coroutine `wait_closed()`

Attends que la méthode `close()` se termine.

Cette méthode est une *coroutine*.

sockets

List of `socket.socket` objects the server is listening to, or `None` if the server is closed.

Handle

class `asyncio.Handle`

A callback wrapper object returned by `AbstractEventLoop.call_soon()`, `AbstractEventLoop.call_soon_threadsafe()`, `AbstractEventLoop.call_later()`, and `AbstractEventLoop.call_at()`.

cancel()

Cancel the call. If the callback is already canceled or executed, this method has no effect.

Exemples de boucles d'évènements

« Hello World » avec `call_soon()`

Example using the `AbstractEventLoop.call_soon()` method to schedule a callback. The callback displays "Hello World" and then stops the event loop :

```
import asyncio

def hello_world(loop):
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
loop.run_forever()
loop.close()
```

Voir aussi :

The *Hello World coroutine* example uses a *coroutine*.

Afficher la date actuelle avec `call_later()`

Example of callback displaying the current date every second. The callback uses the `AbstractEventLoop.call_later()` method to reschedule itself during 5 seconds, and then stops the event loop :

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
loop.run_forever()
loop.close()
```

Voir aussi :

The *coroutine displaying the current date* example uses a *coroutine*.

Watch a file descriptor for read events

Wait until a file descriptor received some data using the `AbstractEventLoop.add_reader()` method and then close the event loop :

```
import asyncio
try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()
loop = asyncio.get_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())
    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)
    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
```

(suite sur la page suivante)

(suite de la page précédente)

```

loop.call_soon(wsock.send, 'abc'.encode())

# Run the event loop
loop.run_forever()

# We are done, close sockets and the event loop
rsock.close()
wsock.close()
loop.close()

```

Voir aussi :

The *register an open socket to wait for data using a protocol* example uses a low-level protocol created by the `AbstractEventLoop.create_connection()` method.

The *register an open socket to wait for data using streams* example uses high-level streams created by the `open_connection()` function in a coroutine.

Définit les gestionnaires de signaux pour *SIGINT* et *SIGTERM*

Register handlers for signals `SIGINT` and `SIGTERM` using the `AbstractEventLoop.add_signal_handler()` method :

```

import asyncio
import functools
import os
import signal

def ask_exit(signame):
    print("got signal %s: exit" % signame)
    loop.stop()

loop = asyncio.get_event_loop()
for signame in ('SIGINT', 'SIGTERM'):
    loop.add_signal_handler(getattr(signal, signame),
                           functools.partial(ask_exit, signame))

print("Event loop running forever, press Ctrl+C to interrupt.")
print("pid %s: send SIGINT or SIGTERM to exit." % os.getpid())
try:
    loop.run_forever()
finally:
    loop.close()

```

Cet exemple fonctionne seulement sur Unix.

18.5.2 Boucles d'évènements

Source code : [Lib/asyncio/events.py](#)

Fonctions des boucles d'évènements

The following functions are convenient shortcuts to accessing the methods of the global policy. Note that this provides access to the default policy, unless an alternative policy was set by calling `set_event_loop_policy()` earlier in the execution of the process.

```
asyncio.get_event_loop()
    Equivalent to calling get_event_loop_policy().get_event_loop().

asyncio.set_event_loop(loop)
    Equivalent to calling get_event_loop_policy().set_event_loop(loop).

asyncio.new_event_loop()
    Equivalent to calling get_event_loop_policy().new_event_loop().
```

Boucles d'évènements disponibles

asyncio currently provides two implementations of event loops : *SelectorEventLoop* and *ProactorEventLoop*.

class `asyncio.SelectorEventLoop`

Event loop based on the *selectors* module. Subclass of *AbstractEventLoop*.

Use the most efficient selector available on the platform.

On Windows, only sockets are supported (ex : pipes are not supported) : see the [MSDN documentation of select](#).

class `asyncio.ProactorEventLoop`

Proactor event loop for Windows using « I/O Completion Ports » aka IOCP. Subclass of *AbstractEventLoop*.

Disponibilité : Windows.

Voir aussi :

[MSDN documentation on I/O Completion Ports](#).

Example to use a *ProactorEventLoop* on Windows :

```
import asyncio, sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

Support des plateformes

The *asyncio* module has been designed to be portable, but each platform still has subtle differences and may not support all *asyncio* features.

Windows

Common limits of Windows event loops :

- `create_unix_connection()` and `create_unix_server()` are not supported : the socket family `socket.AF_UNIX` is specific to UNIX
- `add_signal_handler()` and `remove_signal_handler()` are not supported
- `EventLoopPolicy.set_child_watcher()` is not supported. `ProactorEventLoop` supports sub-processes. It has only one implementation to watch child processes, there is no need to configure it.

Limites spécifiques à `SelectorEventLoop` :

- `SelectSelector` is used which only supports sockets and is limited to 512 sockets.
- `add_reader()` and `add_writer()` only accept file descriptors of sockets
- Pipes are not supported (ex : `connect_read_pipe()`, `connect_write_pipe()`)
- `Subprocesses` are not supported (ex : `subprocess_exec()`, `subprocess_shell()`)

Limites spécifiques à `ProactorEventLoop` :

- `create_datagram_endpoint()` (UDP) is not supported
- `add_reader()` and `add_writer()` are not supported

The resolution of the monotonic clock on Windows is usually around 15.6 msec. The best resolution is 0.5 msec. The resolution depends on the hardware (availability of HPET) and on the Windows configuration. See [asyncio delayed calls](#).

Modifié dans la version 3.5 : La classe `ProactorEventLoop` gère maintenant le SSL.

Mac OS X

Character devices like PTY are only well supported since Mavericks (Mac OS 10.9). They are not supported at all on Mac OS 10.5 and older.

On Mac OS 10.6, 10.7 and 10.8, the default event loop is `SelectorEventLoop` which uses `selectors.KqueueSelector`. `selectors.KqueueSelector` does not support character devices on these versions. The `SelectorEventLoop` can be used with `SelectSelector` or `PollSelector` to support character devices on these versions of Mac OS X. Example :

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

Event loop policies and the default policy

Event loop management is abstracted with a *policy* pattern, to provide maximal flexibility for custom platforms and frameworks. Throughout the execution of a process, a single global policy object manages the event loops available to the process based on the calling context. A policy is an object implementing the `AbstractEventLoopPolicy` interface.

For most users of `asyncio`, policies never have to be dealt with explicitly, since the default global policy is sufficient (see below).

The module-level functions `get_event_loop()` and `set_event_loop()` provide convenient access to event loops managed by the default policy.

Event loop policy interface

An event loop policy must implement the following interface :

class `asyncio.AbstractEventLoopPolicy`

Event loop policy.

get_event_loop()

Récupère la boucle d'évènements pour le contexte actuel.

Returns an event loop object implementing the `AbstractEventLoop` interface. In case called from co-routine, it returns the currently running event loop.

Raises an exception in case no event loop has been set for the current context and the current policy does not specify to create one. It must never return `None`.

Modifié dans la version 3.6.

set_event_loop(loop)

Définit la boucle d'évènements du contexte actuel sur *loop*.

new_event_loop()

Create and return a new event loop object according to this policy's rules.

If there's need to set this loop as the event loop for the current context, `set_event_loop()` must be called explicitly.

The default policy defines context as the current thread, and manages an event loop per thread that interacts with `asyncio`. If the current thread doesn't already have an event loop associated with it, the default policy's `get_event_loop()` method creates one when called from the main thread, but raises `RuntimeError` otherwise.

Access to the global loop policy

`asyncio.get_event_loop_policy()`

Get the current event loop policy.

`asyncio.set_event_loop_policy(policy)`

Set the current event loop policy. If *policy* is `None`, the default policy is restored.

Customizing the event loop policy

To implement a new event loop policy, it is recommended you subclass the concrete default event loop policy `DefaultEventLoopPolicy` and override the methods for which you want to change behavior, for example :

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

18.5.3 Tâches et coroutines

Source code : [Lib/asyncio/tasks.py](#)

Source code : [Lib/asyncio/coroutines.py](#)

Coroutines

Coroutines used with `asyncio` may be implemented using the `async def` statement, or by using *generators*. The `async def` type of coroutine was added in Python 3.5, and is recommended if there is no need to support older Python versions.

Generator-based coroutines should be decorated with `@asyncio.coroutine`, although this is not strictly enforced. The decorator enables compatibility with `async def` coroutines, and also serves as documentation. Generator-based coroutines use the `yield from` syntax introduced in [PEP 380](#), instead of the original `yield` syntax.

The word « coroutine », like the word « generator », is used for two different (though related) concepts :

- The function that defines a coroutine (a function definition using `async def` or decorated with `@asyncio.coroutine`). If disambiguation is needed we will call this a *coroutine function* (`iscoroutinefunction()` returns True).
- The object obtained by calling a coroutine function. This object represents a computation or an I/O operation (usually a combination) that will complete eventually. If disambiguation is needed we will call it a *coroutine object* (`iscoroutine()` returns True).

Les choses que les coroutines peuvent faire :

- `result = await future` or `result = yield from future` – suspends the coroutine until the future is done, then returns the future's result, or raises an exception, which will be propagated. (If the future is cancelled, it will raise a `CancelledError` exception.) Note that tasks are futures, and everything said about futures also applies to tasks.
- `result = await coroutine` or `result = yield from coroutine` – wait for another coroutine to produce a result (or raise an exception, which will be propagated). The `coroutine` expression must be a *call* to another coroutine.
- `return expression` – produce a result to the coroutine that is waiting for this one using `await` or `yield from`.
- `raise exception` – raise an exception in the coroutine that is waiting for this one using `await` or `yield from`.

Calling a coroutine does not start its code running – the coroutine object returned by the call doesn't do anything until you schedule its execution. There are two basic ways to start it running : call `await coroutine` or `yield from coroutine` from another coroutine (assuming the other coroutine is already running !), or schedule its execution using the `ensure_future()` function or the `AbstractEventLoop.create_task()` method.

Coroutines (and tasks) can only run when the event loop is running.

`@asyncio.coroutine`

Decorator to mark generator-based coroutines. This enables the generator use `yield from` to call `async def` coroutines, and also enables the generator to be called by `async def` coroutines, for instance using an `await` expression.

There is no need to decorate `async def` coroutines themselves.

If the generator is not yielded from before it is destroyed, an error message is logged. See [Detect coroutines never scheduled](#).

Note : In this documentation, some methods are documented as coroutines, even if they are plain Python functions returning a `Future`. This is intentional to have a freedom of tweaking the implementation of these functions in the future. If such a function is needed to be used in a callback-style code, wrap its result with `ensure_future()`.

Exemple : Coroutine « Hello World »

Example of coroutine displaying "Hello World" :

```
import asyncio

async def hello_world():
    print("Hello World!")

loop = asyncio.get_event_loop()
# Blocking call which returns when the hello_world() coroutine is done
loop.run_until_complete(hello_world())
loop.close()
```

Voir aussi :

The *Hello World with call_soon()* example uses the `AbstractEventLoop.call_soon()` method to schedule a callback.

Exemple : Coroutine affichant la date actuelle

Example of coroutine displaying the current date every second during 5 seconds using the `sleep()` function :

```
import asyncio
import datetime

async def display_date(loop):
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()
# Blocking call which returns when the display_date() coroutine is done
loop.run_until_complete(display_date(loop))
loop.close()
```

Voir aussi :

The *display the current date with call_later()* example uses a callback with the `AbstractEventLoop.call_later()` method.

Exemple : Chaîner des coroutines

Example chaining coroutines :

```
import asyncio

async def compute(x, y):
    print("Compute %s + %s ..." % (x, y))
    await asyncio.sleep(1.0)
    return x + y
```

(suite sur la page suivante)

(suite de la page précédente)

```

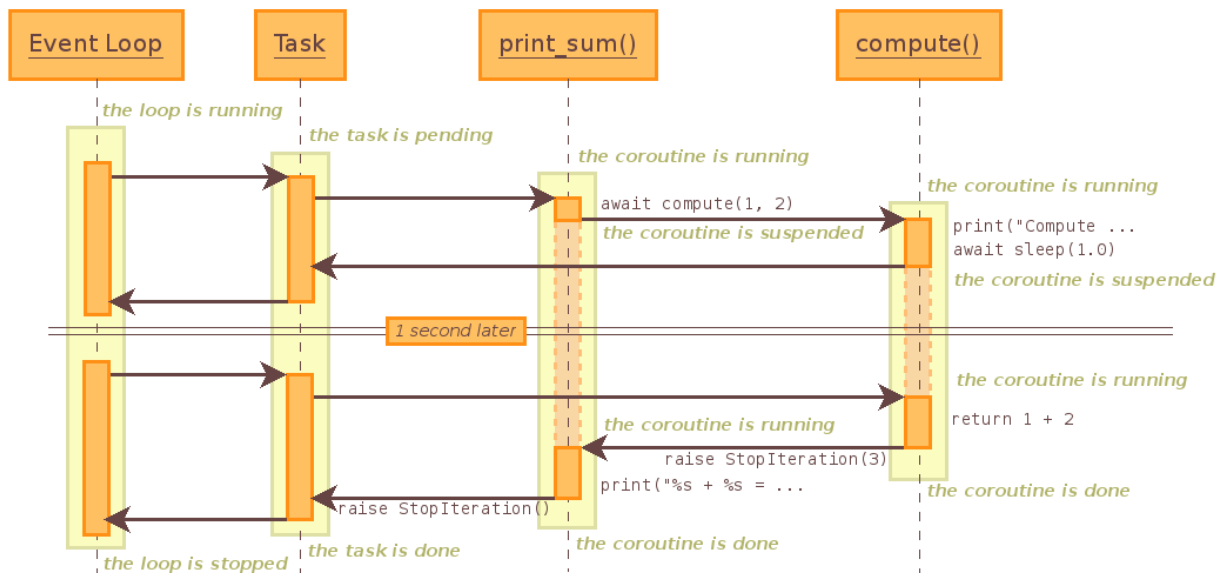
async def print_sum(x, y):
    result = await compute(x, y)
    print("%s + %s = %s" % (x, y, result))

loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()

```

`compute()` is chained to `print_sum()` : `print_sum()` coroutine waits until `compute()` is completed before returning its result.

Sequence diagram of the example :



The « Task » is created by the `AbstractEventLoop.run_until_complete()` method when it gets a coroutine object instead of a task.

The diagram shows the control flow, it does not describe exactly how things work internally. For example, the sleep coroutine creates an internal future which uses `AbstractEventLoop.call_later()` to wake up the task in 1 second.

InvalidStateError

exception `asyncio.InvalidStateError`
L'opération n'est pas autorisée dans cet état.

TimeoutError

exception `asyncio.TimeoutError`
L'opération a dépassé le délai donné.

Note : This exception is different from the builtin `TimeoutError` exception !

Future

class `asyncio.Future` (*, loop=None)
This class is *almost* compatible with `concurrent.futures.Future`.
Différences :
— `result()` and `exception()` do not take a timeout argument and raise an exception when the future isn't done yet.
— Callbacks registered with `add_done_callback()` are always called via the event loop's `call_soon()`.
— This class is not compatible with the `wait()` and `as_completed()` functions in the `concurrent.futures` package.
This class is *not thread safe*.
cancel()
Cancel the future and schedule callbacks.
If the future is already done or cancelled, return `False`. Otherwise, change the future's state to cancelled, schedule the callbacks and return `True`.
cancelled()
Return `True` if the future was cancelled.
done()
Return `True` if the future is done.
Done means either that a result / exception are available, or that the future was cancelled.
result()
Return the result this future represents.
If the future has been cancelled, raises `CancelledError`. If the future's result isn't yet available, raises `InvalidStateError`. If the future is done and has an exception set, this exception is raised.
exception()
Return the exception that was set on this future.
The exception (or `None` if no exception was set) is returned only if the future is done. If the future has been cancelled, raises `CancelledError`. If the future isn't done yet, raises `InvalidStateError`.
add_done_callback(fn)
Add a callback to be run when the future becomes done.
The callback is called with a single argument - the future object. If the future is already done when this is called, the callback is scheduled with `call_soon()`.
Use `functools.partial` to pass parameters to the callback. For example, `fut.add_done_callback(functools.partial(print, "Future:", flush=True))` will call `print("Future:", fut, flush=True)`.

remove_done_callback (*fn*)

Remove all instances of a callback from the « call when done » list.

Donne le nombre de fonctions de rappel supprimées.

set_result (*result*)

Marque le futur comme terminé et définit son résultat.

If the future is already done when this method is called, raises *InvalidStateError*.

set_exception (*exception*)

Marque le futur comme terminé et définit une exception.

If the future is already done when this method is called, raises *InvalidStateError*.

Exemple : Futur avec run_until_complete()

Example combining a *Future* and a *coroutine function* :

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
loop.run_until_complete(future)
print(future.result())
loop.close()
```

The coroutine function is responsible for the computation (which takes 1 second) and it stores the result into the future. The *run_until_complete()* method waits for the completion of the future.

Note : The *run_until_complete()* method uses internally the *add_done_callback()* method to be notified when the future is done.

Exemple : Futur avec run_forever()

The previous example can be written differently using the *Future.add_done_callback()* method to describe explicitly the control flow :

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

def got_result(future):
    print(future.result())
    loop.stop()

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
```

(suite sur la page suivante)

(suite de la page précédente)

```
future.add_done_callback(got_result)
try:
    loop.run_forever()
finally:
    loop.close()
```

In this example, the future is used to link `slow_operation()` to `got_result()` : when `slow_operation()` is done, `got_result()` is called with the result.

Task

class `asyncio.Task` (*coro*, *, *loop=None*)

Schedule the execution of a *coroutine* : wrap it in a future. A task is a subclass of *Future*.

A task is responsible for executing a coroutine object in an event loop. If the wrapped coroutine yields from a future, the task suspends the execution of the wrapped coroutine and waits for the completion of the future. When the future is done, the execution of the wrapped coroutine restarts with the result or the exception of the future.

Event loops use cooperative scheduling : an event loop only runs one task at a time. Other tasks may run in parallel if other event loops are running in different threads. While a task waits for the completion of a future, the event loop executes a new task.

The cancellation of a task is different from the cancelation of a future. Calling `cancel()` will throw a *CancelledError* to the wrapped coroutine. `cancelled()` only returns `True` if the wrapped coroutine did not catch the *CancelledError* exception, or raised a *CancelledError* exception.

If a pending task is destroyed, the execution of its wrapped *coroutine* did not complete. It is probably a bug and a warning is logged : see *Pending task destroyed*.

Don't directly create *Task* instances : use the `ensure_future()` function or the *AbstractEventLoop.create_task()* method.

This class is *not thread safe*.

classmethod `all_tasks` (*loop=None*)

Renvoie l'ensemble des tâches d'une boucle d'évènements.

By default all tasks for the current event loop are returned.

classmethod `current_task` (*loop=None*)

Return the currently running task in an event loop or `None`.

By default the current task for the current event loop is returned.

`None` is returned when called not in the context of a *Task*.

cancel ()

Demande à ce qu'une tâche s'annule elle même.

This arranges for a *CancelledError* to be thrown into the wrapped coroutine on the next cycle through the event loop. The coroutine then has a chance to clean up or even deny the request using `try/except/finally`.

Unlike *Future.cancel()*, this does not guarantee that the task will be cancelled : the exception might be caught and acted upon, delaying cancellation of the task or preventing cancellation completely. The task may also return a value or raise a different exception.

Immediately after this method is called, `cancelled()` will not return `True` (unless the task was already cancelled). A task will be marked as cancelled when the wrapped coroutine terminates with a *CancelledError* exception (even if `cancel()` was not called).

get_stack (*, *limit=None*)

Return the list of stack frames for this task's coroutine.

If the coroutine is not done, this returns the stack where it is suspended. If the coroutine has completed successfully or was cancelled, this returns an empty list. If the coroutine was terminated by an exception, this returns the list of traceback frames.

La pile est toujours affichée de l'appelant à l'appelé.

The optional limit gives the maximum number of frames to return ; by default all available frames are returned. Its meaning differs depending on whether a stack or a traceback is returned : the newest frames of a stack are returned, but the oldest frames of a traceback are returned. (This matches the behavior of the traceback module.)

For reasons beyond our control, only one stack frame is returned for a suspended coroutine.

print_stack (*, limit=None, file=None)

Print the stack or traceback for this task's coroutine.

This produces output similar to that of the traceback module, for the frames retrieved by get_stack(). The limit argument is passed to get_stack(). The file argument is an I/O stream to which the output is written ; by default output is written to sys.stderr.

Exemple : Exécution parallèle de tâches

Exemple d'exécution de trois tâches (A, B, C) en parallèle :

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number+1):
        print("Task %s: Compute factorial(%s)..." % (name, i))
        await asyncio.sleep(1)
        f *= i
    print("Task %s: factorial(%s) = %s" % (name, number, f))

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.gather(
    factorial("A", 2),
    factorial("B", 3),
    factorial("C", 4),
))
loop.close()
```

Sortie :

```
Task A: Compute factorial(2)...
Task B: Compute factorial(2)...
Task C: Compute factorial(2)...
Task A: factorial(2) = 2
Task B: Compute factorial(3)...
Task C: Compute factorial(3)...
Task B: factorial(3) = 6
Task C: Compute factorial(4)...
Task C: factorial(4) = 24
```

A task is automatically scheduled for execution when it is created. The event loop stops when all tasks are done.

Task functions

Note : In the functions below, the optional *loop* argument allows explicitly setting the event loop object used by the underlying task or coroutine. If it's not provided, the default event loop is used.

`asyncio.as_completed(fs, *, loop=None, timeout=None)`

Return an iterator whose values, when waited for, are *Future* instances.

Lève une exception `asyncio.TimeoutError` si le délai d'attente est dépassé avant que tous les futurs ne soient achevés.

Exemple :

```
for f in as_completed(fs):
    result = yield from f # The 'yield from' may raise
    # Use result
```

Note : The futures *f* are not necessarily members of *fs*.

`asyncio.ensure_future(coro_or_future, *, loop=None)`

Schedule the execution of a *coroutine object* : wrap it in a future. Return a *Task* object.

If the argument is a *Future*, it is returned directly.

Nouveau dans la version 3.4.4.

Modifié dans la version 3.5.1 : La fonction accepte n'importe quel objet *awaitable*.

Voir aussi :

The `AbstractEventLoop.create_task()` method.

`asyncio.async(coro_or_future, *, loop=None)`

Un alias obsolète de `ensure_future()`.

Obsolète depuis la version 3.4.4.

`asyncio.wrap_future(future, *, loop=None)`

Wrap a `concurrent.futures.Future` object in a *Future* object.

`asyncio.gather(*coros_or_futures, loop=None, return_exceptions=False)`

Return a future aggregating results from the given coroutine objects or futures.

All futures must share the same event loop. If all the tasks are done successfully, the returned future's result is the list of results (in the order of the original sequence, not necessarily the order of results arrival). If *return_exceptions* is true, exceptions in the tasks are treated the same as successful results, and gathered in the result list ; otherwise, the first raised exception will be immediately propagated to the returned future.

Cancellation : if the outer Future is cancelled, all children (that have not completed yet) are also cancelled. If any child is cancelled, this is treated as if it raised `CancelledError` – the outer Future is *not* cancelled in this case. (This is to prevent the cancellation of one child to cause other children to be cancelled.)

Modifié dans la version 3.6.6 : Si *gather* est lui-même annulé, l'annulation est propagée indépendamment de *return_exceptions*.

`asyncio.iscoroutine(obj)`

Return True if *obj* is a *coroutine object*, which may be based on a generator or an `async def` coroutine.

`asyncio.iscoroutinefunction(func)`

Return True if *func* is determined to be a *coroutine function*, which may be a decorated generator function or an `async def` function.

`asyncio.run_coroutine_threadsafe(coro, loop)`

Submit a *coroutine object* to a given event loop.

Return a `concurrent.futures.Future` to access the result.

This function is meant to be called from a different thread than the one where the event loop is running. Usage :

```
# Create a coroutine
coro = asyncio.sleep(1, result=3)
# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)
# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

If an exception is raised in the coroutine, the returned future will be notified. It can also be used to cancel the task in the event loop :

```
try:
    result = future.result(timeout)
except asyncio.TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print('The coroutine raised an exception: {!r}'.format(exc))
else:
    print('The coroutine returned: {!r}'.format(result))
```

Voir la section *exécution concurrente et multi-fils d'exécution* de la documentation.

Note : Unlike other functions from the module, `run_coroutine_threadsafe()` requires the `loop` argument to be passed explicitly.

Nouveau dans la version 3.5.1.

coroutine `asyncio.sleep(delay, result=None, *, loop=None)`

Create a *coroutine* that completes after a given time (in seconds). If `result` is provided, it is produced to the caller when the coroutine completes.

The resolution of the sleep depends on the *granularity of the event loop*.

Cette fonction est une *coroutine*.

coroutine `asyncio.shield(arg, *, loop=None)`

Attends un future, en le protégeant des annulations.

L'instruction :

```
res = yield from shield(something())
```

est exactement équivalente à l'instruction :

```
res = yield from something()
```

except that if the coroutine containing it is cancelled, the task running in `something()` is not cancelled. From the point of view of `something()`, the cancellation did not happen. But its caller is still cancelled, so the `yield-from` expression still raises `CancelledError`. Note : If `something()` is cancelled by other means this will still cancel `shield()`.

If you want to completely ignore cancellation (not recommended) you can combine `shield()` with a `try/except` clause, as follows :

```
try:
    res = yield from shield(something())
except CancelledError:
    res = None
```

coroutine `asyncio.wait(futures, *, loop=None, timeout=None, return_when=ALL_COMPLETED)`

Wait for the Futures and coroutine objects given by the sequence *futures* to complete. Coroutines will be wrapped in Tasks. Returns two sets of *Future*: (done, pending).

The sequence *futures* must not be empty.

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or *None*, there is no limit to the wait time.

return_when indicates when this function should return. It must be one of the following constants of the `concurrent.futures` module :

Constante	Description
<code>FIRST_COMPLETED</code>	La fonction se termine lorsque n'importe quel futur se termine ou est annulé.
<code>FIRST_EXCEPTION</code>	La fonction se termine lorsque n'importe quel futur se termine en levant une exception. Si aucun <i>futur</i> ne lève d'exception, équivaut à <code>ALL_COMPLETED</code> .
<code>ALL_COMPLETED</code>	La fonction se termine lorsque les <i>futurs</i> sont tous finis ou annulés.

Cette fonction est une *coroutine*.

Utilisation :

```
done, pending = yield from asyncio.wait(fs)
```

Note : This does not raise `asyncio.TimeoutError`! Futures that aren't done when the timeout occurs are returned in the second set.

coroutine `asyncio.wait_for(fut, timeout, *, loop=None)`

Wait for the single *Future* or *coroutine object* to complete with timeout. If *timeout* is *None*, block until the future completes.

Coroutine will be wrapped in *Task*.

Returns result of the Future or coroutine. When a timeout occurs, it cancels the task and raises `asyncio.TimeoutError`. To avoid the task cancellation, wrap it in `shield()`.

If the wait is cancelled, the future *fut* is also cancelled.

Cette fonction est une *coroutine*, utilisation :

```
result = yield from asyncio.wait_for(fut, 60.0)
```

Modifié dans la version 3.4.3 : If the wait is cancelled, the future *fut* is now also cancelled.

18.5.4 Transports et protocoles (API basée sur des fonctions de rappel)

Source code : [Lib/asyncio/transports.py](#)

Source code : [Lib/asyncio/protocols.py](#)

Transports

Transports are classes provided by `asyncio` in order to abstract various kinds of communication channels. You generally won't instantiate a transport yourself ; instead, you will call an `AbstractEventLoop` method which will create the transport and try to initiate the underlying communication channel, calling you back when it succeeds.

Once the communication channel is established, a transport is always paired with a *protocol* instance. The protocol can then call the transport's methods for various purposes.

`asyncio` currently implements transports for TCP, UDP, SSL, and subprocess pipes. The methods available on a transport depend on the transport's kind.

The transport classes are *not thread safe*.

Modifié dans la version 3.6 : The socket option `TCP_NODELAY` is now set by default.

BaseTransport

class `asyncio.BaseTransport`

Classe de base des transports.

close()

Close the transport. If the transport has a buffer for outgoing data, buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's `connection_lost()` method will be called with *None* as its argument.

is_closing()

Return True if the transport is closing or is closed.

Nouveau dans la version 3.5.1.

get_extra_info(name, default=None)

Return optional transport information. *name* is a string representing the piece of transport-specific information to get, *default* is the value to return if the information doesn't exist.

This method allows transport implementations to easily expose channel-specific information.

— socket :

— 'peername' : the remote address to which the socket is connected, result of `socket.socket.getpeername()` (None on error)

— 'socket' : Instance de `socket.socket`

— 'sockname' : the socket's own address, result of `socket.socket.getsockname()`

— Connecteur (socket en anglais) SSL :

— 'compression' : the compression algorithm being used as a string, or None if the connection isn't compressed; result of `ssl.SSLSocket.compression()`

— 'cipher' : a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used; result of `ssl.SSLSocket.cipher()`

— 'peercert' : peer certificate; result of `ssl.SSLSocket.getpeercert()`

— 'sslcontext' : Instance de `ssl.SSLContext`

— 'ssl_object' : `ssl.SSLObject` or `ssl.SSLSocket` instance

— pipe :

— 'pipe' : objet *pipe*

— sous-processus :

— 'subprocess' : `subprocess.Popen` instance

set_protocol(protocol)

Set a new protocol. Switching protocol should only be done when both protocols are documented to support the switch.

Nouveau dans la version 3.5.3.

get_protocol()

Renvoie le protocole courant.

Nouveau dans la version 3.5.3.

Modifié dans la version 3.5.1 : 'ssl_object' est ajouté aux *sockets* SSL.

ReadTransport

class `asyncio.ReadTransport`

Interface pour les transports en lecture seule.

pause_reading()

Pause the receiving end of the transport. No data will be passed to the protocol's `data_received()` method until `resume_reading()` is called.

Modifié dans la version 3.6.7 : The method is idempotent, i.e. it can be called when the transport is already paused or closed.

resume_reading()

Resume the receiving end. The protocol's `data_received()` method will be called once again if some data is available for reading.

Modifié dans la version 3.6.7 : The method is idempotent, i.e. it can be called when the transport is already reading.

WriteTransport

class `asyncio.WriteTransport`

Interface pour les transports en écriture seule.

abort()

Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's `connection_lost()` method will eventually be called with `None` as its argument.

can_write_eof()

Return `True` if the transport supports `write_eof()`, `False` if not.

get_write_buffer_size()

Return the current size of the output buffer used by the transport.

get_write_buffer_limits()

Get the *high*- and *low*-water limits for write flow control. Return a tuple (*low*, *high*) where *low* and *high* are positive number of bytes.

Use `set_write_buffer_limits()` to set the limits.

Nouveau dans la version 3.4.2.

set_write_buffer_limits (*high=None*, *low=None*)

Set the *high*- and *low*-water limits for write flow control.

These two values (measured in number of bytes) control when the protocol's `pause_writing()` and `resume_writing()` methods are called. If specified, the low-water limit must be less than or equal to the high-water limit. Neither *high* nor *low* can be negative.

`pause_writing()` is called when the buffer size becomes greater than or equal to the *high* value. If writing has been paused, `resume_writing()` is called when the buffer size becomes less than or equal to the *low* value.

The defaults are implementation-specific. If only the high-water limit is given, the low-water limit defaults to an implementation-specific value less than or equal to the high-water limit. Setting *high* to zero forces *low* to zero as well, and causes `pause_writing()` to be called whenever the buffer becomes non-empty. Setting *low* to zero causes `resume_writing()` to be called only once the buffer is empty. Use of zero for either limit is generally sub-optimal as it reduces opportunities for doing I/O and computation concurrently.

Use `get_write_buffer_limits()` to get the limits.

write (*data*)

Écrit des octets de *data* sur le transport.

This method does not block ; it buffers the data and arranges for it to be sent out asynchronously.

writelines (*list_of_data*)

Write a list (or any iterable) of data bytes to the transport. This is functionally equivalent to calling *write()* on each element yielded by the iterable, but may be implemented more efficiently.

write_eof ()

Close the write end of the transport after flushing buffered data. Data may still be received.

This method can raise *NotImplementedError* if the transport (e.g. SSL) doesn't support half-closes.

DatagramTransport

DatagramTransport.**sendto** (*data*, *addr=None*)

Send the *data* bytes to the remote peer given by *addr* (a transport-dependent target address). If *addr* is *None*, the data is sent to the target address given on transport creation.

This method does not block ; it buffers the data and arranges for it to be sent out asynchronously.

DatagramTransport.**abort** ()

Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's *connection_lost()* method will eventually be called with *None* as its argument.

BaseSubprocessTransport

class *asyncio*.**BaseSubprocessTransport**

get_pid ()

Donne l'identifiant du sous processus sous la forme d'un nombre entier.

get_pipe_transport (*fd*)

Return the transport for the communication pipe corresponding to the integer file descriptor *fd* :

- 0 : readable streaming transport of the standard input (*stdin*), or *None* if the subprocess was not created with *stdin=PIPE*
- 1 : writable streaming transport of the standard output (*stdout*), or *None* if the subprocess was not created with *stdout=PIPE*
- 2 : writable streaming transport of the standard error (*stderr*), or *None* if the subprocess was not created with *stderr=PIPE*
- autre *fd* : *None*

get_returncode ()

Return the subprocess returncode as an integer or *None* if it hasn't returned, similarly to the *subprocess.Popen.returncode* attribute.

kill ()

Kill the subprocess, as in *subprocess.Popen.kill()*.

On POSIX systems, the function sends SIGKILL to the subprocess. On Windows, this method is an alias for *terminate()*.

send_signal (*signal*)

Send the *signal* number to the subprocess, as in *subprocess.Popen.send_signal()*.

terminate ()

Ask the subprocess to stop, as in *subprocess.Popen.terminate()*. This method is an alias for the *close()* method.

On POSIX systems, this method sends SIGTERM to the subprocess. On Windows, the Windows API function *TerminateProcess()* is called to stop the subprocess.

close ()

Ask the subprocess to stop by calling the *terminate()* method if the subprocess hasn't returned yet, and close transports of all pipes (*stdin*, *stdout* and *stderr*).

Protocols

`asyncio` provides base classes that you can subclass to implement your network protocols. Those classes are used in conjunction with `transports` (see below) : the protocol parses incoming data and asks for the writing of outgoing data, while the transport is responsible for the actual I/O and buffering.

When subclassing a protocol class, it is recommended you override certain methods. Those methods are callbacks : they will be called by the transport on certain events (for example when some data is received) ; you shouldn't call them yourself, unless you are implementing a transport.

Note : All callbacks have default implementations, which are empty. Therefore, you only need to implement the callbacks for the events in which you are interested.

Protocol classes

class `asyncio.Protocol`

The base class for implementing streaming protocols (for use with e.g. TCP and SSL transports).

class `asyncio.DatagramProtocol`

The base class for implementing datagram protocols (for use with e.g. UDP transports).

class `asyncio.SubprocessProtocol`

The base class for implementing protocols communicating with child processes (through a set of unidirectional pipes).

Connection callbacks

These callbacks may be called on `Protocol`, `DatagramProtocol` and `SubprocessProtocol` instances :

`BaseProtocol.connection_made(transport)`

Appelé lorsqu'une connexion est établie.

The `transport` argument is the transport representing the connection. You are responsible for storing it somewhere (e.g. as an attribute) if you need to.

`BaseProtocol.connection_lost(exc)`

Appelé lorsqu'une connexion est perdue ou fermée.

The argument is either an exception object or `None`. The latter means a regular EOF is received, or the connection was aborted or closed by this side of the connection.

`connection_made()` and `connection_lost()` are called exactly once per successful connection. All other callbacks will be called between those two methods, which allows for easier resource management in your protocol implementation.

The following callbacks may be called only on `SubprocessProtocol` instances :

`SubprocessProtocol.pipe_data_received(fd, data)`

Called when the child process writes data into its stdout or stderr pipe. `fd` is the integer file descriptor of the pipe. `data` is a non-empty bytes object containing the data.

`SubprocessProtocol.pipe_connection_lost(fd, exc)`

Called when one of the pipes communicating with the child process is closed. `fd` is the integer file descriptor that was closed.

`SubprocessProtocol.process_exited()`

Appelé lorsqu'un processus enfant se termine.

Streaming protocols

The following callbacks are called on *Protocol* instances :

`Protocol.data_received(data)`

Called when some data is received. *data* is a non-empty bytes object containing the incoming data.

Note : Whether the data is buffered, chunked or reassembled depends on the transport. In general, you shouldn't rely on specific semantics and instead make your parsing generic and flexible enough. However, data is always received in the correct order.

`Protocol.eof_received()`

Called when the other end signals it won't send any more data (for example by calling `write_eof()`, if the other end also uses `asyncio`).

This method may return a false value (including `None`), in which case the transport will close itself. Conversely, if this method returns a true value, closing the transport is up to the protocol. Since the default implementation returns `None`, it implicitly closes the connection.

Note : Some transports such as SSL don't support half-closed connections, in which case returning true from this method will not prevent closing the connection.

`data_received()` can be called an arbitrary number of times during a connection. However, `eof_received()` is called at most once and, if called, `data_received()` won't be called after it.

Machine à états :

```
start -> connection_made() [-> data_received() *] [-> eof_received() ?] ->
connection_lost() -> end
```

Protocoles de datagrammes

The following callbacks are called on *DatagramProtocol* instances.

`DatagramProtocol.datagram_received(data, addr)`

Called when a datagram is received. *data* is a bytes object containing the incoming data. *addr* is the address of the peer sending the data ; the exact format depends on the transport.

`DatagramProtocol.error_received(exc)`

Called when a previous send or receive operation raises an *OSError*. *exc* is the *OSError* instance.

This method is called in rare conditions, when the transport (e.g. UDP) detects that a datagram couldn't be delivered to its recipient. In many conditions though, undeliverable datagrams will be silently dropped.

Flow control callbacks

These callbacks may be called on *Protocol*, *DatagramProtocol* and *SubprocessProtocol* instances :

`BaseProtocol.pause_writing()`

Called when the transport's buffer goes over the high-water mark.

`BaseProtocol.resume_writing()`

Called when the transport's buffer drains below the low-water mark.

`pause_writing()` and `resume_writing()` calls are paired – `pause_writing()` is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually `resume_writing()` is called once when the buffer size reaches the low-water mark.

Note : If the buffer size equals the high-water mark, `pause_writing()` is not called – it must go strictly over. Conversely, `resume_writing()` is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

Note : On BSD systems (OS X, FreeBSD, etc.) flow control is not supported for *DatagramProtocol*, because send failures caused by writing too many packets cannot be detected easily. The socket always appears “ready” and excess packets are dropped; an *OSError* with `errno` set to `errno.ENOBUFS` may or may not be raised; if it is raised, it will be reported to *DatagramProtocol.error_received()* but otherwise ignored.

Coroutines et protocoles

Coroutines can be scheduled in a protocol method using *ensure_future()*, but there is no guarantee made about the execution order. Protocols are not aware of coroutines created in protocol methods and so will not wait for them.

To have a reliable execution order, use *stream objects* in a coroutine with `yield from`. For example, the *StreamWriter.drain()* coroutine can be used to wait until the write buffer is flushed.

Exemples de protocole

Protocole « *echo client* » en TCP

TCP echo client using the *AbstractEventLoop.create_connection()* method, send data and wait until the connection is closed :

```
import asyncio

class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        print('Stop the event loop')
        self.loop.stop()

loop = asyncio.get_event_loop()
message = 'Hello World!'
coro = loop.create_connection(lambda: EchoClientProtocol(message, loop),
                              '127.0.0.1', 8888)
```

(suite sur la page suivante)

(suite de la page précédente)

```

loop.run_until_complete(coro)
loop.run_forever()
loop.close()

```

The event loop is running twice. The `run_until_complete()` method is preferred in this short example to raise an exception if the server is not listening, instead of having to write a short coroutine to handle the exception and stop the running loop. At `run_until_complete()` exit, the loop is no longer running, so there is no need to stop the loop in case of an error.

Voir aussi :

The *TCP echo client using streams* example uses the `asyncio.open_connection()` function.

Protocole « *echo serveur* » en TCP

TCP echo server using the `AbstractEventLoop.create_server()` method, send back received data and close the connection :

```

import asyncio

class EchoServerClientProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()

loop = asyncio.get_event_loop()
# Each client connection will create a new protocol instance
coro = loop.create_server(EchoServerClientProtocol, '127.0.0.1', 8888)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()

```

`Transport.close()` can be called immediately after `WriteTransport.write()` even if data are not sent yet on the socket : both methods are asynchronous. `yield from` is not needed because these transport methods are not coroutines.

Voir aussi :

The *TCP echo server using streams* example uses the `asyncio.start_server()` function.

Protocole « *echo client* » en UDP

UDP echo client using the `AbstractEventLoop.create_datagram_endpoint()` method, send data and close the transport when we received the answer :

```
import asyncio

class EchoClientProtocol:
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Socket closed, stop the event loop")
        loop = asyncio.get_event_loop()
        loop.stop()

loop = asyncio.get_event_loop()
message = "Hello World!"
connect = loop.create_datagram_endpoint(
    lambda: EchoClientProtocol(message, loop),
    remote_addr=('127.0.0.1', 9999))
transport, protocol = loop.run_until_complete(connect)
loop.run_forever()
transport.close()
loop.close()
```


Protocole « *echo serveur* » en UDP

UDP echo server using the `AbstractEventLoop.create_datagram_endpoint()` method, send back received data :

```
import asyncio

class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)

loop = asyncio.get_event_loop()
print("Starting UDP server")
# One protocol instance will be created to serve all client requests
listen = loop.create_datagram_endpoint(
    EchoServerProtocol, local_addr=('127.0.0.1', 9999))
transport, protocol = loop.run_until_complete(listen)

try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

transport.close()
loop.close()
```

Register an open socket to wait for data using a protocol

Wait until a socket receives data using the `AbstractEventLoop.create_connection()` method with a protocol, and then close the event loop

```
import asyncio
try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair

# Create a pair of connected sockets
rsock, wsock = socketpair()
loop = asyncio.get_event_loop()

class MyProtocol(asyncio.Protocol):
    transport = None

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())
```

(suite sur la page suivante)

(suite de la page précédente)

```
# We are done: close the transport (it will call connection_lost())
self.transport.close()

def connection_lost(self, exc):
    # The socket has been closed, stop the event loop
    loop.stop()

# Register the socket to wait for data
connect_coro = loop.create_connection(MyProtocol, sock=rsock)
transport, protocol = loop.run_until_complete(connect_coro)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

# Run the event loop
loop.run_forever()

# We are done, close sockets and the event loop
rsock.close()
wsock.close()
loop.close()
```

Voir aussi :

The *watch a file descriptor for read events* example uses the low-level `AbstractEventLoop.add_reader()` method to register the file descriptor of a socket.

The *register an open socket to wait for data using streams* example uses high-level streams created by the `open_connection()` function in a coroutine.

18.5.5 Streams (coroutine based API)

Source code : `Lib/asyncio/streams.py`

Stream functions

Note : The top-level functions in this module are meant as convenience wrappers only; there's really nothing special there, and if they don't do exactly what you want, feel free to copy their code.

coroutine `asyncio.open_connection` (*host=None, port=None, *, loop=None, limit=None, **kws*)

A wrapper for `create_connection()` returning a (reader, writer) pair.

The reader returned is a `StreamReader` instance; the writer is a `StreamWriter` instance.

The arguments are all the usual arguments to `AbstractEventLoop.create_connection()` except `protocol_factory`; most common are positional `host` and `port`, with various optional keyword arguments following.

Additional optional keyword arguments are `loop` (to set the event loop instance to use) and `limit` (to set the buffer limit passed to the `StreamReader`).

Cette fonction est une *coroutine*.

coroutine `asyncio.start_server` (*client_connected_cb, host=None, port=None, *, loop=None, limit=None, **kws*)

Start a socket server, with a callback for each client connected. The return value is the same as `create_server()`.

The `client_connected_cb` parameter is called with two parameters : `client_reader`, `client_writer`. `client_reader` is a `StreamReader` object, while `client_writer` is a `StreamWriter` object. The `client_connected_cb` parameter can either be a plain callback function or a *coroutine function*; if it is a coroutine function, it will be automatically converted into a *Task*.

The rest of the arguments are all the usual arguments to `create_server()` except `protocol_factory`; most common are positional `host` and `port`, with various optional keyword arguments following.

Additional optional keyword arguments are `loop` (to set the event loop instance to use) and `limit` (to set the buffer limit passed to the `StreamReader`).

Cette fonction est une *coroutine*.

coroutine `asyncio.open_unix_connection` (`path=None`, *, `loop=None`, `limit=None`, ****kws**)

A wrapper for `create_unix_connection()` returning a (reader, writer) pair.

See `open_connection()` for information about return value and other details.

Cette fonction est une *coroutine*.

Disponible sur : UNIX.

coroutine `asyncio.start_unix_server` (`client_connected_cb`, `path=None`, *, `loop=None`, `limit=None`, ****kws**)

Start a UNIX Domain Socket server, with a callback for each client connected.

See `start_server()` for information about return value and other details.

Cette fonction est une *coroutine*.

Disponible sur : UNIX.

StreamReader

class `asyncio.StreamReader` (`limit=_DEFAULT_LIMIT`, `loop=None`)

This class is *not thread safe*.

The `limit` argument's default value is set to `_DEFAULT_LIMIT` which is `2**16` (64 KiB)

exception ()

Récupère l'exception.

feed_eof ()

Acknowledge the EOF.

feed_data (`data`)

Feed `data` bytes in the internal buffer. Any operations waiting for the data will be resumed.

set_exception (`exc`)

Set the exception.

set_transport (`transport`)

Set the transport.

coroutine read (`n=-1`)

Read up to `n` bytes. If `n` is not provided, or set to `-1`, read until EOF and return all read bytes.

If the EOF was received and the internal buffer is empty, return an empty `bytes` object.

Cette méthode est une *coroutine*.

coroutine readline ()

Read one line, where « line » is a sequence of bytes ending with `\n`.

If EOF is received, and `\n` was not found, the method will return the partial read bytes.

If the EOF was received and the internal buffer is empty, return an empty `bytes` object.

Cette méthode est une *coroutine*.

coroutine readexactly (`n`)

Read exactly `n` bytes. Raise an `IncompleteReadError` if the end of the stream is reached before `n` can be read, the `IncompleteReadError.partial` attribute of the exception contains the partial read bytes.

Cette méthode est une *coroutine*.

coroutine readuntil (*separator=b'\n'*)

Read data from the stream until *separator* is found.

On success, the data and separator will be removed from the internal buffer (consumed). Returned data will include the separator at the end.

Configured stream limit is used to check result. Limit sets the maximal length of data that can be returned, not counting the separator.

If an EOF occurs and the complete separator is still not found, an *IncompleteReadError* exception will be raised, and the internal buffer will be reset. The *IncompleteReadError.partial* attribute may contain the separator partially.

If the data cannot be read because of over limit, a *LimitOverrunError* exception will be raised, and the data will be left in the internal buffer, so it can be read again.

Nouveau dans la version 3.5.2.

at_eof ()

Return True if the buffer is empty and *feed_eof()* was called.

StreamWriter

class `asyncio.StreamWriter` (*transport, protocol, reader, loop*)

Wraps a Transport.

This exposes *write()*, *writelines()*, *can_write_eof()*, *write_eof()*, *get_extra_info()* and *close()*. It adds *drain()* which returns an optional *Future* on which you can wait for flow control. It also adds a transport attribute which references the Transport directly.

This class is *not thread safe*.

transport

Transport.

can_write_eof ()

Return *True* if the transport supports *write_eof()*, *False* if not. See *WriteTransport.can_write_eof()*.

close ()

Close the transport : see *BaseTransport.close()*.

coroutine drain ()

Let the write buffer of the underlying transport a chance to be flushed.

The intended use is to write :

```
w.write(data)
yield from w.drain()
```

When the size of the transport buffer reaches the high-water limit (the protocol is paused), block until the size of the buffer is drained down to the low-water limit and the protocol is resumed. When there is nothing to wait for, the yield-from continues immediately.

Yielding from *drain()* gives the opportunity for the loop to schedule the write operation and flush the buffer. It should especially be used when a possibly large amount of data is written to the transport, and the coroutine does not yield-from between calls to *write()*.

Cette méthode est une *coroutine*.

get_extra_info (*name, default=None*)

Return optional transport information : see *BaseTransport.get_extra_info()*.

write (*data*)

Write some *data* bytes to the transport : see *WriteTransport.write()*.

writelines (*data*)

Write a list (or any iterable) of data bytes to the transport : see *WriteTransport.writelines()*.

write_eof ()

Close the write end of the transport after flushing buffered data : see *WriteTransport.write_eof()*.

StreamReaderProtocol

class `asyncio.StreamReaderProtocol` (*stream_reader*, *client_connected_cb*=None, *loop*=None)

Trivial helper class to adapt between *Protocol* and *StreamReader*. Subclass of *Protocol*.

stream_reader is a *StreamReader* instance, *client_connected_cb* is an optional function called with (stream_reader, stream_writer) when a connection is made, *loop* is the event loop instance to use.

(This is a helper class instead of making *StreamReader* itself a *Protocol* subclass, because the *StreamReader* has other potential uses, and to prevent the user of the *StreamReader* from accidentally calling inappropriate methods of the protocol.)

IncompleteReadError

exception `asyncio.IncompleteReadError`

Incomplete read error, subclass of *EOFError*.

expected

Nombre total d'octets attendus (*int*).

partial

Read bytes string before the end of stream was reached (*bytes*).

LimitOverrunError

exception `asyncio.LimitOverrunError`

Reached the buffer limit while looking for a separator.

consumed

Total number of to be consumed bytes.

Stream examples

TCP echo client using streams

TCP echo client using the `asyncio.open_connection()` function :

```
import asyncio

@asyncio.coroutine
def tcp_echo_client(message, loop):
    reader, writer = yield from asyncio.open_connection('127.0.0.1', 8888,
                                                         loop=loop)

    print('Send: %r' % message)
    writer.write(message.encode())

    data = yield from reader.read(100)
    print('Received: %r' % data.decode())

    print('Close the socket')
    writer.close()

message = 'Hello World!'
loop = asyncio.get_event_loop()
loop.run_until_complete(tcp_echo_client(message, loop))
loop.close()
```

Voir aussi :

The *TCP echo client protocol* example uses the `AbstractEventLoop.create_connection()` method.

TCP echo server using streams

TCP echo server using the `asyncio.start_server()` function :

```
import asyncio

@asyncio.coroutine
def handle_echo(reader, writer):
    data = yield from reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')
    print("Received %r from %r" % (message, addr))

    print("Send: %r" % message)
    writer.write(data)
    yield from writer.drain()

    print("Close the client socket")
    writer.close()

loop = asyncio.get_event_loop()
coro = asyncio.start_server(handle_echo, '127.0.0.1', 8888, loop=loop)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

Voir aussi :

The *TCP echo server protocol* example uses the `AbstractEventLoop.create_server()` method.

Récupère les en-têtes HTTP

Simple example querying HTTP headers of the URL passed on the command line :

```
import asyncio
import urllib.parse
import sys

@asyncio.coroutine
def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
```

(suite sur la page suivante)

(suite de la page précédente)

```

        connect = asyncio.open_connection(url.hostname, 443, ssl=True)
    else:
        connect = asyncio.open_connection(url.hostname, 80)
    reader, writer = yield from connect
    query = ('HEAD {path} HTTP/1.0\r\n'
            'Host: {hostname}\r\n'
            '\r\n').format(path=url.path or '/', hostname=url.hostname)
    writer.write(query.encode('latin-1'))
    while True:
        line = yield from reader.readline()
        if not line:
            break
        line = line.decode('latin1').rstrip()
        if line:
            print('HTTP header> %s' % line)

    # Ignore the body, close the socket
    writer.close()

url = sys.argv[1]
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(print_http_headers(url))
loop.run_until_complete(task)
loop.close()

```

Utilisation :

```
python example.py http://example.com/path/page.html
```

ou avec HTTPS :

```
python example.py https://example.com/path/page.html
```

Register an open socket to wait for data using streams

Coroutine waiting until a socket receives data using the `open_connection()` function :

```

import asyncio
try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair

@asyncio.coroutine
def wait_for_data(loop):
    # Create a pair of connected sockets
    rsock, wsock = socketpair()

    # Register the open socket to wait for data
    reader, writer = yield from asyncio.open_connection(sock=rsock, loop=loop)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data

```

(suite sur la page suivante)

(suite de la page précédente)

```
data = yield from reader.read(100)

# Got data, we are done: close the socket
print("Received:", data.decode())
writer.close()

# Close the second socket
wsock.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(wait_for_data(loop))
loop.close()
```

Voir aussi :

The *register an open socket to wait for data using a protocol* example uses a low-level protocol created by the `AbstractEventLoop.create_connection()` method.

The *watch a file descriptor for read events* example uses the low-level `AbstractEventLoop.add_reader()` method to register the file descriptor of a socket.

18.5.6 Subprocess

Source code : `Lib/asyncio/subprocess.py`

Boucle d'évènements Windows

On Windows, the default event loop is `SelectorEventLoop` which does not support subprocesses. `ProactorEventLoop` should be used instead. Example to use it on Windows :

```
import asyncio, sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

Voir aussi :

Available event loops and *Platform support*.

Créer un processus fils : API de haut niveau utilisant `Process`

coroutine `asyncio.create_subprocess_exec(*args, stdin=None, stdout=None, stderr=None, loop=None, limit=None, **kwargs)`

Create a subprocess.

The *limit* parameter sets the buffer limit passed to the `StreamReader`. See `AbstractEventLoop.subprocess_exec()` for other parameters.

Return a `Process` instance.

Cette fonction est une *coroutine*.

coroutine `asyncio.create_subprocess_shell(cmd, stdin=None, stdout=None, stderr=None, loop=None, limit=None, **kwargs)`

Exécute la commande *shell cmd*.

The *limit* parameter sets the buffer limit passed to the *StreamReader*. See *AbstractEventLoop.subprocess_shell()* for other parameters.

Return a *Process* instance.

It is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid *shell injection* vulnerabilities. The *shlex.quote()* function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

Cette fonction est une *coroutine*.

Use the *AbstractEventLoop.connect_read_pipe()* and *AbstractEventLoop.connect_write_pipe()* methods to connect pipes.

Create a subprocess : low-level API using *subprocess.Popen*

Run subprocesses asynchronously using the *subprocess* module.

coroutine *AbstractEventLoop.subprocess_exec* (*protocol_factory*, **args*, *st-*
din=subprocess.PIPE, stdout=subprocess.PIPE,
*stderr=subprocess.PIPE, **kwargs*)

Create a subprocess from one or more string arguments (character strings or bytes strings encoded to the *filesystem encoding*), where the first string specifies the program to execute, and the remaining strings specify the program's arguments. (Thus, together the string arguments form the *sys.argv* value of the program, assuming it is a Python script.) This is similar to the standard library *subprocess.Popen* class called with *shell=False* and the list of strings passed as the first argument; however, where *Popen* takes a single argument which is list of strings, *subprocess_exec()* takes multiple string arguments.

The *protocol_factory* must instantiate a subclass of the *asyncio.SubprocessProtocol* class.

Autres paramètres :

- *stdin* : Either a file-like object representing the pipe to be connected to the subprocess's standard input stream using *connect_write_pipe()*, or the constant *subprocess.PIPE* (the default). By default a new pipe will be created and connected.
- *stdout* : Either a file-like object representing the pipe to be connected to the subprocess's standard output stream using *connect_read_pipe()*, or the constant *subprocess.PIPE* (the default). By default a new pipe will be created and connected.
- *stderr* : Either a file-like object representing the pipe to be connected to the subprocess's standard error stream using *connect_read_pipe()*, or one of the constants *subprocess.PIPE* (the default) or *subprocess.STDOUT*. By default a new pipe will be created and connected. When *subprocess.STDOUT* is specified, the subprocess's standard error stream will be connected to the same pipe as the standard output stream.
- All other keyword arguments are passed to *subprocess.Popen* without interpretation, except for *bufsize*, *universal_newlines* and *shell*, which should not be specified at all.

Returns a pair of (transport, protocol), where *transport* is an instance of *BaseSubprocessTransport*.

Cette méthode est une *coroutine*.

See the constructor of the *subprocess.Popen* class for parameters.

coroutine *AbstractEventLoop.subprocess_shell* (*protocol_factory*, *cmd*, ***, *st-*
din=subprocess.PIPE, *st-*
dout=subprocess.PIPE, *st-*
*derr=subprocess.PIPE, **kwargs*)

Create a subprocess from *cmd*, which is a character string or a bytes string encoded to the *filesystem encoding*, using the platform's « shell » syntax. This is similar to the standard library *subprocess.Popen* class called with *shell=True*.

The *protocol_factory* must instantiate a subclass of the *asyncio.SubprocessProtocol* class.

See *subprocess_exec()* for more details about the remaining arguments.

Returns a pair of `(transport, protocol)`, where *transport* is an instance of `BaseSubprocessTransport`.

It is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid [shell injection](#) vulnerabilities. The `shlex.quote()` function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

Cette méthode est une *coroutine*.

Voir aussi :

The `AbstractEventLoop.connect_read_pipe()` and `AbstractEventLoop.connect_write_pipe()` methods.

Constantes

`asyncio.subprocess.PIPE`

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `create_subprocess_shell()` and `create_subprocess_exec()` and indicates that a pipe to the standard stream should be opened.

`asyncio.subprocess.STDOUT`

Special value that can be used as the *stderr* argument to `create_subprocess_shell()` and `create_subprocess_exec()` and indicates that standard error should go into the same handle as standard output.

`asyncio.subprocess.DEVNULL`

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `create_subprocess_shell()` and `create_subprocess_exec()` and indicates that the special file `os.devnull` will be used.

Process

class `asyncio.subprocess.Process`

A subprocess created by the `create_subprocess_exec()` or the `create_subprocess_shell()` function.

The API of the `Process` class was designed to be close to the API of the `subprocess.Popen` class, but there are some differences :

- There is no explicit `poll()` method
- The `communicate()` and `wait()` methods don't take a *timeout* parameter : use the `wait_for()` function
- The *universal_newlines* parameter is not supported (only bytes strings are supported)
- The `wait()` method of the `Process` class is asynchronous whereas the `wait()` method of the `Popen` class is implemented as a busy loop.

This class is *not thread safe*. See also the *Subprocess and threads* section.

coroutine `wait()`

Wait for child process to terminate. Set and return *returncode* attribute.

Cette méthode est une *coroutine*.

Note : This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use the `communicate()` method when using pipes to avoid that.

coroutine `communicate(input=None)`

Interact with process : Send data to *stdin*. Read data from *stdout* and *stderr*, until end-of-file is reached. Wait for process to terminate. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. The type of *input* must be bytes.

`communicate()` returns a tuple `(stdout_data, stderr_data)`.

If a `BrokenPipeError` or `ConnectionResetError` exception is raised when writing *input* into `stdin`, the exception is ignored. It occurs when the process exits before all data are written into `stdin`.

Note that if you want to send data to the process's `stdin`, you need to create the `Process` object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

Cette méthode est une *coroutine*.

Note : Les données lues sont mises en cache en mémoire, donc n'utilisez pas cette méthode si la taille des données est importante voire illimitée.

Modifié dans la version 3.4.2 : The method now ignores `BrokenPipeError` and `ConnectionResetError`.

send_signal (*signal*)

Envoie le signal *signal* au sous-processus.

Note : On Windows, `SIGTERM` is an alias for `terminate()`. `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

terminate ()

Stop the child. On Posix OSs the method sends `signal.SIGTERM` to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

kill ()

Kills the child. On Posix OSs the function sends `SIGKILL` to the child. On Windows `kill()` is an alias for `terminate()`.

stdin

Standard input stream (*StreamWriter*), `None` if the process was created with `stdin=None`.

stdout

Standard output stream (*StreamReader*), `None` if the process was created with `stdout=None`.

stderr

Standard error stream (*StreamReader*), `None` if the process was created with `stderr=None`.

Avertissement : Use the `communicate()` method rather than `.stdin.write`, `.stdout.read` or `.stderr.read` to avoid deadlocks due to streams pausing reading or writing and blocking the child process.

pid

L'identifiant du processus.

Note that for processes created by the `create_subprocess_shell()` function, this attribute is the process identifier of the spawned shell.

returncode

Return code of the process when it exited. A `None` value indicates that the process has not terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (Unix only).

Processus fils et fils d'exécution

`asyncio` supports running subprocesses from different threads, but there are limits :

- Une boucle d'évènements doit être exécutée sur le fil d'exécution principal
- The child watcher must be instantiated in the main thread, before executing subprocesses from other threads. Call the `get_child_watcher()` function in the main thread to instantiate the child watcher.

The `asyncio.subprocess.Process` class is not thread safe.

Voir aussi :

The *Concurrency and multithreading in asyncio* section.

Subprocess examples

Subprocess using transport and protocol

Example of a subprocess protocol using to get the output of a subprocess and to wait for the subprocess exit. The subprocess is created by the `AbstractEventLoop.subprocess_exec()` method :

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

@asyncio.coroutine
def get_date(loop):
    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by the protocol DateProtocol,
    # redirect the standard output into a pipe
    create = loop.subprocess_exec(lambda: DateProtocol(exit_future),
                                  sys.executable, '-c', code,
                                  stdin=None, stderr=None)

    transport, protocol = yield from create

    # Wait for the subprocess exit using the process_exited() method
    # of the protocol
    yield from exit_future

    # Close the stdout pipe
    transport.close()

    # Read the output which was collected by the pipe_data_received()
    # method of the protocol
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()
```

(suite sur la page suivante)

(suite de la page précédente)

```

if sys.platform == "win32":
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
else:
    loop = asyncio.get_event_loop()

date = loop.run_until_complete(get_date(loop))
print("Current date: %s" % date)
loop.close()

```

Subprocess using streams

Example using the *Process* class to control the subprocess and the *StreamReader* class to read from the standard output. The subprocess is created by the *create_subprocess_exec()* function :

```

import asyncio.subprocess
import sys

@asyncio.coroutine
def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess, redirect the standard output into a pipe
    create = asyncio.create_subprocess_exec(sys.executable, '-c', code,
                                           stdout=asyncio.subprocess.PIPE)

    proc = yield from create

    # Read one line of output
    data = yield from proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit
    yield from proc.wait()
    return line

if sys.platform == "win32":
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
else:
    loop = asyncio.get_event_loop()

date = loop.run_until_complete(get_date())
print("Current date: %s" % date)
loop.close()

```

18.5.7 Primitives de synchronisation

Source code : [Lib/asyncio/locks.py](#)

Locks :

- *Lock*
- *Event*
- *Condition*

Sémaphores :

- *Semaphore*
- *BoundedSemaphore*

asyncio lock API was designed to be close to classes of the *threading* module (*Lock*, *Event*, *Condition*, *Semaphore*, *BoundedSemaphore*), but it has no *timeout* parameter. The *asyncio.wait_for()* function can be used to cancel a task after a timeout.

Locks

Lock

class `asyncio.Lock` (*, *loop=None*)

Primitive lock objects.

A primitive lock is a synchronization primitive that is not owned by a particular coroutine when locked. A primitive lock is in one of two states, “locked” or “unlocked”.

It is created in the unlocked state. It has two basic methods, *acquire()* and *release()*. When the state is unlocked, *acquire()* changes the state to locked and returns immediately. When the state is locked, *acquire()* blocks until a call to *release()* in another coroutine changes it to unlocked, then the *acquire()* call resets it to locked and returns. The *release()* method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a *RuntimeError* will be raised.

When more than one coroutine is blocked in *acquire()* waiting for the state to turn to unlocked, only one coroutine proceeds when a *release()* call resets the state to unlocked; first coroutine which is blocked in *acquire()* is being processed.

acquire() is a coroutine and should be called with *yield from*.

Locks also support the context management protocol. (*yield from lock*) should be used as the context manager expression.

This class is *not thread safe*.

Utilisation :

```
lock = Lock()
...
yield from lock
try:
    ...
finally:
    lock.release()
```

Context manager usage :

```
lock = Lock()
...
with (yield from lock):
    ...
```

Lock objects can be tested for locking state :

```

if not lock.locked():
    yield from lock
else:
    # lock is acquired
    ...

```

locked()

Donne True si le verrou est acquis.

coroutine acquire()

Acquire a lock.

This method blocks until the lock is unlocked, then sets it to locked and returns True.

Cette méthode est une *coroutine*.

release()

Libère un verrou.

When the lock is locked, reset it to unlocked, and return. If any other coroutines are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a *RuntimeError* is raised.

Il n'y a pas de valeur de retour.

Event

class `asyncio.Event` (*, loop=None)

An Event implementation, asynchronous equivalent to *threading.Event*.

Class implementing event objects. An event manages a flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true. The flag is initially false.

This class is *not thread safe*.

clear()

Reset the internal flag to false. Subsequently, coroutines calling *wait()* will block until *set()* is called to set the internal flag to true again.

is_set()

Return True if and only if the internal flag is true.

set()

Set the internal flag to true. All coroutines waiting for it to become true are awakened. Coroutine that call *wait()* once the flag is true will not block at all.

coroutine wait()

Block until the internal flag is true.

If the internal flag is true on entry, return True immediately. Otherwise, block until another coroutine calls *set()* to set the flag to true, then return True.

Cette méthode est une *coroutine*.

Condition

class `asyncio.Condition` (lock=None, *, loop=None)

A Condition implementation, asynchronous equivalent to *threading.Condition*.

This class implements condition variable objects. A condition variable allows one or more coroutines to wait until they are notified by another coroutine.

If the *lock* argument is given and not None, it must be a *Lock* object, and it is used as the underlying lock. Otherwise, a new *Lock* object is created and used as the underlying lock.

This class is *not thread safe*.

coroutine acquire()

Acquire the underlying lock.

This method blocks until the lock is unlocked, then sets it to locked and returns `True`.

Cette méthode est une *coroutine*.

notify(n=1)

By default, wake up one coroutine waiting on this condition, if any. If the calling coroutine has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method wakes up at most *n* of the coroutines waiting for the condition variable; it is a no-op if no coroutines are waiting.

Note: An awakened coroutine does not actually return from its `wait()` call until it can reacquire the lock. Since `notify()` does not release the lock, its caller should.

locked()

Return `True` if the underlying lock is acquired.

notify_all()

Wake up all coroutines waiting on this condition. This method acts like `notify()`, but wakes up all waiting coroutines instead of one. If the calling coroutine has not acquired the lock when this method is called, a `RuntimeError` is raised.

release()

Libère le verrou sous-jacent.

When the lock is locked, reset it to unlocked, and return. If any other coroutines are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a `RuntimeError` is raised.

Il n'y a pas de valeur de retour.

coroutine wait()

Attends d'être notifié.

If the calling coroutine has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notify_all()` call for the same condition variable in another coroutine. Once awakened, it re-acquires the lock and returns `True`.

Cette méthode est une *coroutine*.

coroutine wait_for(predicate)

Attends jusqu'à ce qu'un prédicat devienne vrai.

The predicate should be a callable which result will be interpreted as a boolean value. The final predicate value is the return value.

Cette méthode est une *coroutine*.

Sémaphores

Sémaphore

class `asyncio.Semaphore` (*value=1, *, loop=None*)

A Semaphore implementation.

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some other coroutine calls `release()`.

Semaphores also support the context management protocol.

The optional argument gives the initial value for the internal counter; it defaults to 1. If the value given is less than 0, `ValueError` is raised.

This class is *not thread safe*.

coroutine acquire()

Acquire a semaphore.

If the internal counter is larger than zero on entry, decrement it by one and return `True` immediately. If it is zero on entry, block, waiting until some other coroutine has called `release()` to make it larger than 0, and then return `True`.

Cette méthode est une *coroutine*.

locked()

Returns `True` if semaphore can not be acquired immediately.

release()

Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another coroutine is waiting for it to become larger than zero again, wake up that coroutine.

BoundedSemaphore

class `asyncio.BoundedSemaphore` (*value=1, *, loop=None*)

A bounded semaphore implementation. Inherit from `Semaphore`.

This raises `ValueError` in `release()` if it would increase the value above the initial value.

18.5.8 Queues

Source code : `Lib/asyncio/queues.py`

Queues :

- `Queue`
- `PriorityQueue`
- `LifoQueue`

`asyncio` queue API was designed to be close to classes of the `queue` module (`Queue`, `PriorityQueue`, `LifoQueue`), but it has no `timeout` parameter. The `asyncio.wait_for()` function can be used to cancel a task after a timeout.

Queue

class `asyncio.Queue` (*maxsize=0, *, loop=None*)

A queue, useful for coordinating producer and consumer coroutines.

If *maxsize* is less than or equal to zero, the queue size is infinite. If it is an integer greater than 0, then `yield` from `put()` will block when the queue reaches *maxsize*, until an item is removed by `get()`.

Unlike the standard library `queue`, you can reliably know this `Queue`'s size with `qsize()`, since your single-threaded `asyncio` application won't be interrupted between calling `qsize()` and doing an operation on the `Queue`.

This class is *not thread safe*.

Modifié dans la version 3.4.4 : Les nouvelles méthodes `join()` et `task_done()`.

empty()

Renvoie `True` si la queue est vide, `False` sinon.

full()

Return `True` if there are *maxsize* items in the queue.

Note : If the `Queue` was initialized with *maxsize*=0 (the default), then `full()` is never `True`.

coroutine get()

Remove and return an item from the queue. If queue is empty, wait until an item is available.

Cette méthode est une *coroutine*.

Voir aussi :

La méthode `empty()`.

get_nowait()

Supprime et donne un élément de la queue.

Return an item if one is immediately available, else raise `QueueEmpty`.

coroutine join()

Bloque jusqu'à ce que tous les éléments de la queue aient été récupérés et traités.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

Cette méthode est une *coroutine*.

Nouveau dans la version 3.4.4.

coroutine put(item)

Put an item into the queue. If the queue is full, wait until a free slot is available before adding item.

Cette méthode est une *coroutine*.

Voir aussi :

La méthode `full()`.

put_nowait(item)

Ajoute un élément dans la queue sans bloquer.

If no free slot is immediately available, raise `QueueFull`.

qsize()

Nombre d'éléments dans la queue.

task_done()

Indicate that a formerly enqueued task is complete.

Used by queue consumers. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.

If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises `ValueError` if called more times than there were items placed in the queue.

Nouveau dans la version 3.4.4.

maxsize

Nombre d'éléments autorisés dans la queue.

PriorityQueue

class `asyncio.PriorityQueue`

A subclass of `Queue`; retrieves entries in priority order (lowest first).

Entries are typically tuples of the form : (priority number, data).

LifoQueue

class `asyncio.LifoQueue`

A subclass of `Queue` that retrieves most recently added entries first.

Exceptions

exception `asyncio.QueueEmpty`

Exception raised when the `get_nowait()` method is called on a `Queue` object which is empty.

exception `asyncio.QueueFull`

Exception raised when the `put_nowait()` method is called on a `Queue` object which is full.

18.5.9 Programmer avec *asyncio*

Asynchronous programming is different than classical « sequential » programming. This page lists common traps and explains how to avoid them.

Mode de débogage d'*asyncio*

The implementation of *asyncio* has been written for performance. In order to ease the development of asynchronous code, you may wish to enable *debug mode*.

Pour activer toutes les vérifications de débogage pour une application :

- Enable the *asyncio* debug mode globally by setting the environment variable `PYTHONASYNCIODEBUG` to 1, or by calling `AbstractEventLoop.set_debug()`.
- Set the log level of the *asyncio* logger to `logging.DEBUG`. For example, call `logging.basicConfig(level=logging.DEBUG)` at startup.
- Configure the `warnings` module to display *ResourceWarning* warnings. For example, use the `-Wdefault` command line option of Python to display them.

Exemples de vérifications de débogage :

- Log *coroutines defined but never « yielded from »*
- `call_soon()` and `call_at()` methods raise an exception if they are called from the wrong thread.
- Enregistre le temps d'exécution du sélecteur dans le journal
- Log callbacks taking more than 100 ms to be executed. The `AbstractEventLoop.slow_callback_duration` attribute is the minimum duration in seconds of « slow » callbacks.
- *ResourceWarning* warnings are emitted when transports and event loops are *not closed explicitly*.

Voir aussi :

The `AbstractEventLoop.set_debug()` method and the *asyncio* logger.

Annulation

Cancellation of tasks is not common in classic programming. In asynchronous programming, not only is it something common, but you have to prepare your code to handle it.

Futures and tasks can be cancelled explicitly with their `Future.cancel()` method. The `wait_for()` function cancels the waited task when the timeout occurs. There are many other cases where a task can be cancelled indirectly.

Don't call `set_result()` or `set_exception()` method of *Future* if the future is cancelled : it would fail with an exception. For example, write :

```
if not fut.cancelled():
    fut.set_result('done')
```

Don't schedule directly a call to the `set_result()` or the `set_exception()` method of a future with `AbstractEventLoop.call_soon()` : the future can be cancelled before its method is called.

If you wait for a future, you should check early if the future was cancelled to avoid useless operations. Example :

```
@coroutine
def slow_operation(fut):
    if fut.cancelled():
        return
    # ... slow computation ...
    yield from fut
    # ...
```

The `shield()` function can also be used to ignore cancellation.

Concurrence et *multithreading*

An event loop runs in a thread and executes all callbacks and tasks in the same thread. While a task is running in the event loop, no other task is running in the same thread. But when the task uses `yield from`, the task is suspended and the event loop executes the next task.

To schedule a callback from a different thread, the `AbstractEventLoop.call_soon_threadsafe()` method should be used. Example :

```
loop.call_soon_threadsafe(callback, *args)
```

Most `asyncio` objects are not thread safe. You should only worry if you access objects outside the event loop. For example, to cancel a future, don't call directly its `Future.cancel()` method, but :

```
loop.call_soon_threadsafe(fut.cancel)
```

To handle signals and to execute subprocesses, the event loop must be run in the main thread.

To schedule a coroutine object from a different thread, the `run_coroutine_threadsafe()` function should be used. It returns a `concurrent.futures.Future` to access the result :

```
future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
result = future.result(timeout) # Wait for the result with a timeout
```

The `AbstractEventLoop.run_in_executor()` method can be used with a thread pool executor to execute a callback in different thread to not block the thread of the event loop.

Voir aussi :

The *Synchronization primitives* section describes ways to synchronize tasks.

The *Subprocess and threads* section lists `asyncio` limitations to run subprocesses from different threads.

Gérer les fonctions bloquantes correctement

Blocking functions should not be called directly. For example, if a function blocks for 1 second, other tasks are delayed by 1 second which can have an important impact on reactivity.

For networking and subprocesses, the `asyncio` module provides high-level APIs like *protocols*.

An executor can be used to run a task in a different thread or even in a different process, to not block the thread of the event loop. See the `AbstractEventLoop.run_in_executor()` method.

Voir aussi :

The *Delayed calls* section details how the event loop handles time.

Journalisation

The `asyncio` module logs information with the `logging` module in the logger `'asyncio'`.

The default log level for the `asyncio` module is `logging.INFO`. For those not wanting such verbosity from `asyncio` the log level can be changed. For example, to change the level to `logging.WARNING`:

```
logging.getLogger('asyncio').setLevel(logging.WARNING)
```

Détecte les coroutines qui ne sont jamais exécutées

When a coroutine function is called and its result is not passed to `ensure_future()` or to the `AbstractEventLoop.create_task()` method, the execution of the coroutine object will never be scheduled which is probably a bug. *Enable the debug mode of asyncio to log a warning* to detect it.

Exemple avec le bug :

```
import asyncio

@asyncio.coroutine
def test():
    print("never scheduled")

test()
```

Affichage en mode débogage :

```
Coroutine test() at test.py:3 was never yielded from
Coroutine object created at (most recent call last):
  File "test.py", line 7, in <module>
    test()
```

The fix is to call the `ensure_future()` function or the `AbstractEventLoop.create_task()` method with the coroutine object.

Voir aussi :

Pending task destroyed.

Detect exceptions never consumed

Python usually calls `sys.excepthook()` on unhandled exceptions. If `Future.set_exception()` is called, but the exception is never consumed, `sys.excepthook()` is not called. Instead, *a log is emitted* when the future is deleted by the garbage collector, with the traceback where the exception was raised.

Example of unhandled exception :

```
import asyncio

@asyncio.coroutine
def bug():
    raise Exception("not consumed")

loop = asyncio.get_event_loop()
asyncio.ensure_future(bug())
loop.run_forever()
loop.close()
```

Sortie :

```
Task exception was never retrieved
future: <Task finished coro=<coro() done, defined at asyncio/coroutines.py:139>_
↳exception=Exception('not consumed',)>
Traceback (most recent call last):
  File "asyncio/tasks.py", line 237, in _step
    result = next(coro)
  File "asyncio/coroutines.py", line 141, in coro
    res = func(*args, **kw)
  File "test.py", line 5, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Enable the debug mode of asyncio to get the traceback where the task was created. Output in debug mode :

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3> exception=Exception(
↳'not consumed',) created at test.py:8>
source_traceback: Object created at (most recent call last):
  File "test.py", line 8, in <module>
    asyncio.ensure_future(bug())
Traceback (most recent call last):
  File "asyncio/tasks.py", line 237, in _step
    result = next(coro)
  File "asyncio/coroutines.py", line 79, in __next__
    return next(self.gen)
  File "asyncio/coroutines.py", line 141, in coro
    res = func(*args, **kw)
  File "test.py", line 5, in bug
    raise Exception("not consumed")
Exception: not consumed
```

There are different options to fix this issue. The first option is to chain the coroutine in another coroutine and use classic try/except :

```
@asyncio.coroutine
def handle_exception():
    try:
        yield from bug()
    except Exception:
        print("exception consumed")

loop = asyncio.get_event_loop()
asyncio.ensure_future(handle_exception())
loop.run_forever()
loop.close()
```

Another option is to use the *AbstractEventLoop.run_until_complete()* function :

```
task = asyncio.ensure_future(bug())
try:
    loop.run_until_complete(task)
except Exception:
    print("exception consumed")
```

Voir aussi :

La méthode *Future.exception()*.

Châîner les coroutines correctement

When a coroutine function calls other coroutine functions and tasks, they should be chained explicitly with `yield from`. Otherwise, the execution is not guaranteed to be sequential.

Example with different bugs using `asyncio.sleep()` to simulate slow operations :

```
import asyncio

@asyncio.coroutine
def create():
    yield from asyncio.sleep(3.0)
    print("(1) create file")

@asyncio.coroutine
def write():
    yield from asyncio.sleep(1.0)
    print("(2) write into file")

@asyncio.coroutine
def close():
    print("(3) close file")

@asyncio.coroutine
def test():
    asyncio.ensure_future(create())
    asyncio.ensure_future(write())
    asyncio.ensure_future(close())
    yield from asyncio.sleep(2.0)
    loop.stop()

loop = asyncio.get_event_loop()
asyncio.ensure_future(test())
loop.run_forever()
print("Pending tasks at exit: %s" % asyncio.Task.all_tasks(loop))
loop.close()
```

Expected output :

```
(1) create file
(2) write into file
(3) close file
Pending tasks at exit: set()
```

Affichage obtenu :

```
(3) close file
(2) write into file
Pending tasks at exit: {<Task pending create() at test.py:7 wait_for=<Future pending_
↳cb=[Task._wakeup()]>>}>
Task was destroyed but it is pending!
task: <Task pending create() done at test.py:5 wait_for=<Future pending cb=[Task._
↳wakeup()]>>
```

The loop stopped before the `create()` finished, `close()` has been called before `write()`, whereas coroutine functions were called in this order : `create()`, `write()`, `close()`.

To fix the example, tasks must be marked with `yield from`:

```
@asyncio.coroutine
def test():
    yield from asyncio.ensure_future(create())
    yield from asyncio.ensure_future(write())
    yield from asyncio.ensure_future(close())
    yield from asyncio.sleep(2.0)
    loop.stop()
```

Ou sans `asyncio.ensure_future()` :

```
@asyncio.coroutine
def test():
    yield from create()
    yield from write()
    yield from close()
    yield from asyncio.sleep(2.0)
    loop.stop()
```

Pending task destroyed

If a pending task is destroyed, the execution of its wrapped *coroutine* did not complete. It is probably a bug and so a warning is logged.

Example of log :

```
Task was destroyed but it is pending!
task: <Task pending coro=<kill_me() done, defined at test.py:5> wait_for=<Future_
↳pending cb=[Task._wakeup()]>>
```

Enable the debug mode of asyncio to get the traceback where the task was created. Example of log in debug mode :

```
Task was destroyed but it is pending!
source_traceback: Object created at (most recent call last):
  File "test.py", line 15, in <module>
    task = asyncio.ensure_future(coro, loop=loop)
task: <Task pending coro=<kill_me() done, defined at test.py:5> wait_for=<Future_
↳pending cb=[Task._wakeup()] created at test.py:7> created at test.py:15>
```

Voir aussi :

Detect coroutine objects never scheduled.

Close transports and event loops

When a transport is no more needed, call its `close()` method to release resources. Event loops must also be closed explicitly.

If a transport or an event loop is not closed explicitly, a *ResourceWarning* warning will be emitted in its destructor. By default, *ResourceWarning* warnings are ignored. The *Debug mode of asyncio* section explains how to display them.

Voir aussi :

Le module *asyncio* a été présenté dans la **PEP 3156**. La **PEP 3153** décrit les motivations premières concernant les couches transports et protocoles.

18.6 `asyncore` — Gestionnaire de socket asynchrone

Code source : [Lib/asyncore.py](#)

Obsolète depuis la version 3.6 : Utilisez `asyncio` à la place.

Note : Ce module n'existe que pour des raisons de rétrocompatibilité. Pour du code nouveau, l'utilisation de `asyncio` est recommandée.

This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do « more than one thing at a time. » Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It's really only practical if your program is largely I/O bound. If your program is processor bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely processor bound, however.

If your operating system supports the `select()` system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the « background. » Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The `asyncore` module solves many of the difficult problems for you, making the task of building sophisticated high-performance network servers and clients a snap. For « conversational » applications and protocols the companion `asynchat` module is invaluable.

The basic idea behind both modules is to create one or more network *channels*, instances of class `asyncore.dispatcher` and `asynchat.async_chat`. Creating the channels adds them to a global map, used by the `loop()` function if you do not provide it with your own *map*.

Once the initial channel(s) is(are) created, calling the `loop()` function activates channel service, which continues until the last channel (including any that have been added to the map during asynchronous service) is closed.

`asyncore.loop([timeout[, use_poll[, map[, count]]]])`

Enter a polling loop that terminates after *count* passes or all open channels have been closed. All arguments are optional. The *count* parameter defaults to `None`, resulting in the loop terminating only when all channels have been closed. The *timeout* argument sets the timeout parameter for the appropriate `select()` or `poll()` call, measured in seconds; the default is 30 seconds. The *use_poll* parameter, if true, indicates that `poll()` should be used in preference to `select()` (the default is `False`).

The *map* parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If *map* is omitted, a global map is used. Channels (instances of `asyncore.dispatcher`, `asynchat.async_chat` and subclasses thereof) can freely be mixed in the map.

class `asyncore.dispatcher`

The `dispatcher` class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are :

Event	Description
<code>handle_connect()</code>	Implied by the first read or write event
<code>handle_close()</code>	Implied by a read event with no data available
<code>handle_accepted()</code>	Implied by a read event on a listening socket

During asynchronous processing, each mapped channel's `readable()` and `writable()` methods are used to determine whether the channel's socket should be added to the list of channels `select()`ed or `poll()`ed for read and write events.

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows :

handle_read()

Called when the asynchronous loop detects that a `read()` call on the channel's socket will succeed.

handle_write()

Called when the asynchronous loop detects that a writable socket can be written. Often this method will implement the necessary buffering for performance. For example :

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()

Called when there is out of band (OOB) data for a socket connection. This will almost never happen, as OOB is tenuously supported and rarely used.

handle_connect()

Called when the active opener's socket actually makes a connection. Might send a « welcome » banner, or initiate a protocol negotiation with the remote endpoint, for example.

handle_close()

Appelé lorsque la socket est fermée.

handle_error()

Called when an exception is raised and not otherwise handled. The default version prints a condensed trace-back.

handle_accept()

Called on listening channels (passive openers) when a connection can be established with a new remote endpoint that has issued a `connect()` call for the local endpoint. Deprecated in version 3.2; use `handle_accepted()` instead.

Obsolète depuis la version 3.2.

handle_accepted(sock, addr)

Called on listening channels (passive openers) when a connection has been established with a new remote endpoint that has issued a `connect()` call for the local endpoint. *sock* is a *new* socket object usable to send and receive data on the connection, and *addr* is the address bound to the socket on the other end of the connection.

Nouveau dans la version 3.2.

readable()

Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which read events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in read events.

writable()

Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which write events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in write events.

In addition, each channel delegates or extends many of the socket methods. Most of these are nearly identical to their socket partners.

create_socket(family=socket.AF_INET, type=socket.SOCK_STREAM)

This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the `socket` documentation for information on creating sockets.

Modifié dans la version 3.3 : Les arguments *family* et *type* sont optionnels.

connect(address)

As with the normal socket object, *address* is a tuple with the first element the host to connect to, and the second the port number.

send (*data*)

Envoie *data* à l'autre bout de la socket.

recv (*buffer_size*)

Read at most *buffer_size* bytes from the socket's remote end-point. An empty bytes object implies that the channel has been closed from the other end.

Note that *recv()* may raise *BlockingIOError*, even though *select.select()* or *select.poll()* has reported the socket ready for reading.

listen (*backlog*)

Listen for connections made to the socket. The *backlog* argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

bind (*address*)

Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — refer to the *socket* documentation for more information.) To mark the socket as re-usable (setting the *SO_REUSEADDR* option), call the *dispatcher* object's *set_reuse_addr()* method.

accept ()

Accept a connection. The socket must be bound to an address and listening for connections. The return value can be either *None* or a pair (*conn*, *address*) where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection. When *None* is returned it means the connection didn't take place, in which case the server should just ignore this event and keep listening for further incoming connections.

close ()

Close the socket. All future operations on the socket object will fail. The remote end-point will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

class *asyncore.dispatcher_with_send*

A *dispatcher* subclass which adds simple buffered output capability, useful for simple clients. For more sophisticated usage use *asynchat.async_chat*.

class *asyncore.file_dispatcher*

A *file_dispatcher* takes a file descriptor or *file object* along with an optional *map* argument and wraps it for use with the *poll()* or *loop()* functions. If provided a file object or anything with a *fileno()* method, that method will be called and passed to the *file_wrapper* constructor. Availability : UNIX.

class *asyncore.file_wrapper*

A *file_wrapper* takes an integer file descriptor and calls *os.dup()* to duplicate the handle so that the original handle may be closed independently of the *file_wrapper*. This class implements sufficient methods to emulate a socket for use by the *file_dispatcher* class. Availability : UNIX.

18.6.1 Exemple de client HTTP basique avec *asyncore*

Here is a very basic HTTP client that uses the *dispatcher* class to implement its socket handling :

```
import asyncore

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect((host, 80))
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                             (path, host), 'ascii')

    def handle_connect(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```

    pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]

client = HTTPClient('www.python.org', '/')
asyncore.loop()

```

18.6.2 Serveur *echo* basique avec *asyncore*

Here is a basic echo server that uses the *dispatcher* class to accept connections and dispatches the incoming connections to a handler :

```

import asyncore

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()

```

18.7 `asyncchat` — Gestionnaire d'interfaces de connexion (*socket*) commande/réponse asynchrones

Code source : `* Lib/asyncchat.py`

Obsolète depuis la version 3.6 : Utilisez `asyncio` à la place.

Note : Ce module n'existe que pour des raisons de rétrocompatibilité. Pour du code nouveau, l'utilisation de `asyncio` est recommandée.

Ce module s'appuie sur l'infrastructure de `asyncore`, en simplifiant les clients et serveurs asynchrones et en rendant plus facile la gestion de protocoles dont les éléments finissent par une chaîne arbitraire, ou sont de longueur variable. `asyncchat` définit une classe abstraite `async_chat` dont vous héritez, et qui fournit des implémentations des méthodes `collect_incoming_data()` et `found_terminator()`. Il utilise la même boucle asynchrone que `asyncore`, et deux types de canaux, `asyncore.dispatcher` et `asyncchat.async_chat`, qui peuvent être librement mélangés dans la carte des canaux. Habituellement, un canal de serveur `asyncore.dispatcher` génère de nouveaux canaux d'objets `asyncchat.async_chat` à la réception de requêtes de connexion.

`class asyncchat.async_chat`

Cette classe est une sous-classe abstraite de `asyncore.dispatcher`. Pour en faire un usage pratique, vous devez créer une classe héritant de `async_chat`, et implémentant des méthodes `collect_incoming_data()` et `found_terminator()` sensées. Les méthodes de `asyncore.dispatcher` peuvent être utilisées, même si toutes n'ont pas de sens dans un contexte de messages/réponse.

Comme `asyncore.dispatcher`, `async_chat` définit un ensemble d'événements générés par une analyse de l'état des interfaces de connexion (*socket* en anglais) après un appel à `select()`. Une fois que la boucle de scrutation (*polling* en anglais) a été lancée, les méthodes des objets `async_chat` sont appelées par le *framework* de traitement d'événements sans que le programmeur n'ait à le spécifier.

Deux attributs de classe peuvent être modifiés, pour améliorer la performance, ou potentiellement pour économiser de la mémoire.

`ac_in_buffer_size`

La taille du tampon d'entrées asynchrones (4096 par défaut).

`ac_out_buffer_size`

La taille du tampon de sorties asynchrones (4096 par défaut).

Contrairement à `asyncore.dispatcher`, `async_chat` permet de définir une queue FIFO de *producteurs*. Un producteur nécessite seulement une méthode, `more()`, qui renvoie la donnée à transmettre au canal. Le producteur indique son épuisement (*c.-à-d.* qu'il ne contient plus de données) en ne retournant avec sa méthode `more()` l'objet `bytes` vide. L'objet `async_chat` retire alors le producteur de la queue et commence à utiliser le producteur suivant, si il y en a un. Quand la queue de producteurs est vide, la méthode `handle_write()` ne fait rien. La méthode `set_terminator()` de l'objet du canal est utilisé pour décrire comment reconnaître la fin, ou la présence d'un point d'arrêt, dans la transmission entrante depuis le point d'accès distant.

Pour construire une sous classe fonctionnelle de `async_chat` pour vos méthodes d'entrées `collect_incoming_data()` et `found_terminator()` doivent gérer la donnée que le canal reçoit de manière asynchrone. Ces méthodes sont décrites ci-dessous.

`async_chat.close_when_done()`

Pousse un `None` sur la pile de producteurs. Quand ce producteur est récupéré dans la queue, le canal est fermé.

`async_chat.collect_incoming_data(data)`

Appelé avec `data` contenant une quantité arbitraire de données. La méthode par défaut, qui doit être écrasée, lève une `NotImplementedError`.

`async_chat.discard_buffers()`

En cas d'urgence, cette méthode va supprimer toute donnée présente dans les tampons d'entrée et/ou de sortie dans la queue de producteurs.

`async_chat.found_terminator()`

Appelée quand le flux de données correspond à la condition de fin décrite par `set_terminator()`. La méthode par défaut, qui doit être écrasée, lève une `NotImplementedError`. Les données entrantes mises en tampon devraient être disponibles via un attribut de l'instance.

`async_chat.get_terminator()`

Renvoie le terminateur courant pour le canal.

`async_chat.push(data)`

Pousse *data* sur la pile du canal pour assurer sa transmission. C'est tout ce dont on a besoin pour que le canal envoie des données sur le réseau. Cependant, il est possible d'utiliser vos propres producteurs dans des schémas plus complexes qui implémentent de la cryptographie et du *chunking* par exemple.

`async_chat.push_with_producer(producer)`

Prends un objet producteur l'ajoute à la queue de producteurs associée au canal. Quand tous les producteurs actuellement poussés ont été épuisés, le canal consomme les données de ce producteur en appelant sa méthode `more()` et envoie les données au point d'accès distant.

`async_chat.set_terminator(term)`

Définit le marqueur de fin que le canal doit reconnaître. *term* peut être n'importe lequel des trois types de valeurs, correspondant aux trois différentes manières de gérer les données entrantes.

<i>term</i>	Description
<i>string</i>	Appellera <code>found_terminator()</code> quand la chaîne est trouvée dans le flux d'entrée
<i>integer</i>	Appellera <code>found_terminator()</code> quand le nombre de caractère indiqué a été reçu
<code>None</code>	Le canal continue de collecter des informations indéfiniment

Notez que toute donnée située après le marqueur de fin sera accessible en lecture par le canal après que `found_terminator()` ait été appelé.

18.7.1 Exemple *asynchat*

L'exemple partiel suivant montre comment des requêtes HTTP peuvent être lues avec *asynchat*. Un serveur web pourrait créer un objet `http_request_handler` pour chaque connexion entrante. Notez que initialement, le marqueur de fin du canal est défini pour reconnaître les lignes vides à la fin des entêtes HTTP, et une option indique que les entêtes sont en train d'être lues.

Une fois que les entêtes ont été lues, si la requête est de type *POST* (ce qui indique que davantage de données sont présentes dans le flux entrant) alors l'entête `Content-Length` est utilisé pour définir un marqueur de fin numérique pour lire la bonne quantité de données depuis le canal.

La méthode `handle_request()` est appelée une fois que toutes les données pertinentes ont été rassemblées, après avoir défini le marqueur de fin à `None` pour s'assurer que toutes les données étrangères envoyées par le client web sont ignorées.

```
import asynchat

class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
```

(suite sur la page suivante)

(suite de la page précédente)

```

self.ibuffer = []
self.obuffer = b""
self.set_terminator(b"\r\n\r\n")
self.reading_headers = True
self.handling = False
self.cgi_data = None
self.log = log

def collect_incoming_data(self, data):
    """Buffer the data"""
    self.ibuffer.append(data)

def found_terminator(self):
    if self.reading_headers:
        self.reading_headers = False
        self.parse_headers(b"".join(self.ibuffer))
        self.ibuffer = []
        if self.op.upper() == b"POST":
            clen = self.headers.getheader("content-length")
            self.set_terminator(int(clen))
        else:
            self.handling = True
            self.set_terminator(None)
            self.handle_request()
    elif not self.handling:
        self.set_terminator(None) # browsers sometimes over-send
        self.cgi_data = parse(self.headers, b"".join(self.ibuffer))
        self.handling = True
        self.ibuffer = []
        self.handle_request()

```

18.8 `signal` — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python.

18.8.1 General rules

The `signal.signal()` function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed : `SIGPIPE` is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and `SIGINT` is translated into a `KeyboardInterrupt` exception.

A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for `SIGCHLD`, which follows the underlying implementation.

Execution of Python signal handlers

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the *virtual machine* to execute the corresponding Python signal handler at a later point (for example at the next *bytecode* instruction). This has consequences :

- It makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV` that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the *faulthandler* module to report on synchronous errors.
- A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when the calculation finishes.

Signals and threads

Python signal handlers are always executed in the main Python thread, even if the signal was received in another thread. This means that signals can't be used as a means of inter-thread communication. You can use the synchronization primitives from the *threading* module instead.

Besides, only the main thread is allowed to set a new signal handler.

18.8.2 Module contents

Modifié dans la version 3.5 : `signal (SIG*)`, handler (`SIG_DFL`, `SIG_IGN`) and sigmask (`SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`) related constants listed below were turned into *enums*. `getsignal()`, `pthread_sigmask()`, `sigpending()` and `sigwait()` functions return human-readable *enums*.

The variables defined in the *signal* module are :

`signal.SIG_DFL`

This is one of two standard signal handling options ; it will simply perform the default function for the signal. For example, on most systems the default action for `SIGQUIT` is to dump core and exit, while the default action for `SIGCHLD` is to simply ignore it.

`signal.SIG_IGN`

This is another standard signal handler, which will simply ignore the given signal.

SIG*

All the signal numbers are defined symbolically. For example, the hangup signal is defined as `signal.SIGHUP` ; the variable names are identical to the names used in C programs, as found in `<signal.h>`. The Unix man page for “`signal()`” lists the existing signals (on some systems this is `signal(2)`, on others the list is in `signal(7)`). Note that not all systems define the same set of signal names ; only those names defined by the system are defined by this module.

`signal.CTRL_C_EVENT`

The signal corresponding to the `Ctrl+C` keystroke event. This signal can only be used with `os.kill()`.

Disponibilité : Windows.

Nouveau dans la version 3.2.

`signal.CTRL_BREAK_EVENT`

The signal corresponding to the `Ctrl+Break` keystroke event. This signal can only be used with `os.kill()`.

Disponibilité : Windows.

Nouveau dans la version 3.2.

`signal.NSIG`

One more than the number of the highest signal number.

`signal.ITIMER_REAL`

Decrements interval timer in real time, and delivers `SIGALRM` upon expiration.

`signal.ITIMER_VIRTUAL`

Decrements interval timer only when the process is executing, and delivers `SIGVTALRM` upon expiration.

`signal.ITIMER_PROF`

Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with `ITIMER_VIRTUAL`, this timer is usually used to profile the time spent by the application in user and kernel space. `SIGPROF` is delivered upon expiration.

`signal.SIG_BLOCK`

A possible value for the *how* parameter to `pthread_sigmask()` indicating that signals are to be blocked. Nouveau dans la version 3.3.

`signal.SIG_UNBLOCK`

A possible value for the *how* parameter to `pthread_sigmask()` indicating that signals are to be unblocked. Nouveau dans la version 3.3.

`signal.SIG_SETMASK`

A possible value for the *how* parameter to `pthread_sigmask()` indicating that the signal mask is to be replaced.

Nouveau dans la version 3.3.

The `signal` module defines one exception :

exception `signal.ItimerError`

Raised to signal an error from the underlying `setitimer()` or `getitimer()` implementation. Expect this error if an invalid interval timer or a negative time is passed to `setitimer()`. This error is a subtype of `OSError`.

Nouveau dans la version 3.3 : This error used to be a subtype of `IOError`, which is now an alias of `OSError`.

The `signal` module defines the following functions :

`signal.alarm(time)`

If *time* is non-zero, this function requests that a `SIGALRM` signal be sent to the process in *time* seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If *time* is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled. (See the Unix man page `alarm(2)`.) Availability : Unix.

`signal.getsignal(signalnum)`

Return the current signal handler for the signal *signalnum*. The returned value may be a callable Python object, or one of the special values `signal.SIG_IGN`, `signal.SIG_DFL` or `None`. Here, `signal.SIG_IGN` means that the signal was previously ignored, `signal.SIG_DFL` means that the default way of handling the signal was previously in use, and `None` means that the previous signal handler was not installed from Python.

`signal.pause()`

Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. Not on Windows. (See the Unix man page `signal(2)`.)

See also `sigwait()`, `sigwaitinfo()`, `sigtimedwait()` and `sigpending()`.

`signal.pthread_kill(thread_id, signalnum)`

Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be *executed by the main thread*. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with `InterruptedError`.

Use `threading.get_ident()` or the *ident* attribute of `threading.Thread` objects to get a suitable value for *thread_id*.

If *signalnum* is 0, then no signal is sent, but error checking is still performed ; this can be used to check if the target thread is still running.

Availability : Unix (see the man page *pthread_kill(3)* for further information).

See also *os.kill()*.

Nouveau dans la version 3.3.

signal.pthread_sigmask (*how*, *mask*)

Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of *how*, as follows.

- *SIG_BLOCK* : The set of blocked signals is the union of the current set and the *mask* argument.
- *SIG_UNBLOCK* : The signals in *mask* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- *SIG_SETMASK* : The set of blocked signals is set to the *mask* argument.

mask is a set of signal numbers (e.g. {*signal.SIGINT*, *signal.SIGTERM*}). Use *range(1, signal.NSIG)* for a full mask including all signals.

For example, *signal.pthread_sigmask(signal.SIG_BLOCK, [])* reads the signal mask of the calling thread.

Availability : Unix. See the man page *sigprocmask(3)* and *pthread_sigmask(3)* for further information.

See also *pause()*, *sigpending()* and *sigwait()*.

Nouveau dans la version 3.3.

signal.setitimer (*which*, *seconds*[, *interval*])

Sets given interval timer (one of *signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* or *signal.ITIMER_PROF*) specified by *which* to fire after *seconds* (float is accepted, different from *alarm()*) and after that every *interval* seconds. The interval timer specified by *which* can be cleared by setting *seconds* to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; *signal.ITIMER_REAL* will deliver *SIGALRM*, *signal.ITIMER_VIRTUAL* sends *SIGVTALRM*, and *signal.ITIMER_PROF* will deliver *SIGPROF*.

The old values are returned as a tuple : (delay, interval).

Attempting to pass an invalid interval timer will cause an *ItimerError*. Availability : Unix.

signal.getitimer (*which*)

Returns current value of a given interval timer specified by *which*. Availability : Unix.

signal.set_wakeup_fd (*fd*)

Set the wakeup file descriptor to *fd*. When a signal is received, the signal number is written as a single byte into the *fd*. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If *fd* is -1, file descriptor wakeup is disabled. If not -1, *fd* must be non-blocking. It is up to the library to remove any bytes from *fd* before calling poll or select again.

Use for example *struct.unpack('%uB' % len(data), data)* to decode the signal numbers list.

When threads are enabled, this function can only be called from the main thread ; attempting to call it from other threads will cause a *ValueError* exception to be raised.

Modifié dans la version 3.5 : On Windows, the function now also supports socket handles.

signal.siginterrupt (*signalnum*, *flag*)

Change system call restart behaviour : if *flag* is *False*, system calls will be restarted when interrupted by signal *signalnum*, otherwise system calls will be interrupted. Returns nothing. Availability : Unix (see the man page *siginterrupt(3)* for further information).

Note that installing a signal handler with *signal()* will reset the restart behaviour to interruptible by implicitly calling *siginterrupt()* with a true *flag* value for the given signal.

`signal.signal(signalnum, handler)`

Set the handler for signal *signalnum* to the function *handler*. *handler* can be a callable Python object taking two arguments (see below), or one of the special values `signal.SIG_IGN` or `signal.SIG_DFL`. The previous signal handler will be returned (see the description of `getsignal()` above). (See the Unix man page `signal(2)`.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a `ValueError` exception to be raised.

The *handler* is called with two arguments: the signal number and the current stack frame (`None` or a frame object; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the `inspect` module).

On Windows, `signal()` can only be called with `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM`, or `SIGBREAK`. A `ValueError` will be raised in any other case. Note that not all systems define the same set of signal names; an `AttributeError` will be raised if a signal name is not defined as `SIG*` module level constant.

`signal.sigpending()`

Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). Return the set of the pending signals.

Availability: Unix (see the man page `sigpending(2)` for further information).

See also `pause()`, `pthread_sigmask()` and `sigwait()`.

Nouveau dans la version 3.3.

`signal.sigwait(sigset)`

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal (removes it from the pending list of signals), and returns the signal number.

Availability: Unix (see the man page `sigwait(3)` for further information).

See also `pause()`, `pthread_sigmask()`, `sigpending()`, `sigwaitinfo()` and `sigtimedwait()`.

Nouveau dans la version 3.3.

`signal.sigwaitinfo(sigset)`

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal and removes it from the pending list of signals. If one of the signals in *sigset* is already pending for the calling thread, the function will return immediately with information about that signal. The signal handler is not called for the delivered signal. The function raises an `InterruptedError` if it is interrupted by a signal that is not in *sigset*.

The return value is an object representing the data contained in the `siginfo_t` structure, namely: `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`.

Availability: Unix (see the man page `sigwaitinfo(2)` for further information).

See also `pause()`, `sigwait()` and `sigtimedwait()`.

Nouveau dans la version 3.3.

Modifié dans la version 3.5: The function is now retried if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see [PEP 475](#) for the rationale).

`signal.sigtimedwait(sigset, timeout)`

Like `sigwaitinfo()`, but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as 0, a poll is performed. Returns `None` if a timeout occurs.

Availability: Unix (see the man page `sigtimedwait(2)` for further information).

See also `pause()`, `sigwait()` and `sigwaitinfo()`.

Nouveau dans la version 3.3.

Modifié dans la version 3.5: The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see [PEP 475](#) for the rationale).

18.8.3 Example

Here is a minimal example program. It uses the `alarm()` function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the `os.open()` to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```
import signal, os

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

18.9 mmap — Memory-mapped file support

Memory-mapped file objects behave like both `bytearray` and like *file objects*. You can use `mmap` objects in most places where `bytearray` are expected; for example, you can use the `re` module to search through a memory-mapped file. You can also change a single byte by doing `obj[index] = 97`, or change a subsequence by assigning to a slice : `obj[i1:i2] = b'...'`. You can also read and write data starting at the current file position, and `seek()` through the file to different positions.

A memory-mapped file is created by the `mmap` constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its `fileno()` method to obtain the correct value for the `fileno` parameter. Otherwise, you can open the file using the `os.open()` function, which returns a file descriptor directly (the file still needs to be closed when done).

Note : If you want to create a memory-mapping for a writable, buffered file, you should `flush()` the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, `access` may be specified as an optional keyword parameter. `access` accepts one of three values : `ACCESS_READ`, `ACCESS_WRITE`, or `ACCESS_COPY` to specify read-only, write-through or copy-on-write memory respectively. `access` can be used on both Unix and Windows. If `access` is not specified, Windows `mmap` returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an `ACCESS_READ` memory map raises a `TypeError` exception. Assignment to an `ACCESS_WRITE` memory map affects both memory and the underlying file. Assignment to an `ACCESS_COPY` memory map affects memory but does not update the underlying file.

To map anonymous memory, -1 should be passed as the `fileno` along with the length.

class `mmap.mmap` (*fileno*, *length*, *tagname*=None, *access*=`ACCESS_DEFAULT`[, *offset*])

(**Windows version**) Maps *length* bytes from the file specified by the file handle *fileno*, and creates a `mmap` object. If *length* is larger than the current size of the file, the file is extended to contain *length* bytes. If *length* is 0, the

maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not `None`, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or `None`, the mapping is created without a name. Avoiding the use of the tag parameter will assist in keeping your code portable between Unix and Windows.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of the `ALLOCATIONGRANULARITY`.

```
class mmap.mmap(fileno, length, flags=MAP_SHARED, prot=PROT_WRITE|PROT_READ, access=ACCESS_DEFAULT[, offset])
```

(Unix version) Maps *length* bytes from the file specified by the file descriptor *fileno*, and returns a mmap object. If *length* is 0, the maximum length of the map will be the current size of the file when *mmap* is called.

flags specifies the nature of the mapping. `MAP_PRIVATE` creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and `MAP_SHARED` creates a mapping that's shared with all other processes mapping the same areas of the file. The default value is `MAP_SHARED`.

prot, if specified, gives the desired memory protection; the two most useful values are `PROT_READ` and `PROT_WRITE`, to specify that the pages may be read or written. *prot* defaults to `PROT_READ | PROT_WRITE`.

access may be specified in lieu of *flags* and *prot* as an optional keyword parameter. It is an error to specify both *flags*, *prot* and *access*. See the description of *access* above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of `ALLOCATIONGRANULARITY` which is equal to `PAGESIZE` on Unix systems.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with physical backing store on Mac OS X and OpenVMS.

This example shows a simple way of using *mmap* :

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline()) # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5]) # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline()) # prints b"Hello world!\n"
    # close the map
    mm.close()
```

mmap can also be used as a context manager in a `with` statement :

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

Nouveau dans la version 3.2 : Context manager support.

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes :

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0: # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

Memory-mapped file objects support the following methods :

close()

Closes the mmap. Subsequent calls to other methods of the object will result in a `ValueError` exception being raised. This will not close the open file.

closed

True if the file is closed.

Nouveau dans la version 3.2.

find(*sub*[, *start*[, *end*]])

Returns the lowest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns -1 on failure.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

flush([*offset*[, *size*])

Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed. *offset* must be a multiple of the `PAGESIZE` or `ALLOCATIONGRANULARITY`.

(Windows version) A nonzero value returned indicates success; zero indicates failure.

(Unix version) A zero value is returned to indicate success. An exception is raised when the call failed.

move(*dest*, *src*, *count*)

Copy the *count* bytes starting at offset *src* to the destination index *dest*. If the mmap was created with `ACCESS_READ`, then calls to move will raise a `TypeError` exception.

read([*n*])

Return a *bytes* containing up to *n* bytes starting from the current file position. If the argument is omitted, `None` or negative, return all bytes from the current file position to the end of the mapping. The file position is updated to point after the bytes that were returned.

Modifié dans la version 3.3 : Argument can be omitted or `None`.

read_byte()

Returns a byte at the current file position as an integer, and advances the file position by 1.

readline()

Returns a single line, starting at the current file position and up to the next newline.

resize(*newsize*)

Resizes the map and the underlying file, if any. If the mmap was created with `ACCESS_READ` or `ACCESS_COPY`, resizing the map will raise a `TypeError` exception.

rfind(*sub*[, *start*[, *end*]])

Returns the highest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns -1 on failure.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

seek (*pos*[, *whence*])

Set the file's current position. *whence* argument is optional and defaults to `os.SEEK_SET` or 0 (absolute file positioning); other values are `os.SEEK_CUR` or 1 (seek relative to the current position) and `os.SEEK_END` or 2 (seek relative to the file's end).

size ()

Return the length of the file, which can be larger than the size of the memory-mapped area.

tell ()

Returns the current position of the file pointer.

write (*bytes*)

Write the bytes in *bytes* into memory at the current position of the file pointer and return the number of bytes written (never less than `len(bytes)`, since if the write fails, a `ValueError` will be raised). The file position is updated to point after the bytes that were written. If the mmap was created with `ACCESS_READ`, then writing to it will raise a `TypeError` exception.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

Modifié dans la version 3.6 : The number of bytes written is now returned.

write_byte (*byte*)

Write the integer *byte* into memory at the current position of the file pointer; the file position is advanced by 1. If the mmap was created with `ACCESS_READ`, then writing to it will raise a `TypeError` exception.

Traitement des données provenant d'Internet

Ce chapitre décrit les modules qui prennent en charge le traitement des formats de données couramment utilisés sur Internet.

19.1 `email` — Un paquet de gestion des e-mails et MIME

Code source : `Lib/email/__init__.py`

Le paquet `email` est une bibliothèque pour gérer les e-mails. Il est spécifiquement conçu pour ne pas gérer les envois d'e-mails vers SMTP ([RFC 2821](#)), NNTP, ou autres serveurs ; ces fonctions sont du ressort des modules comme `smtplib` et `nntplib`. Le paquet `email` tente de respecter les RFC autant que possible, il gère [RFC 5233](#) et [RFC 6532](#), ainsi que les RFCs en rapport avec les MIME comme [RFC 2045](#), [RFC 2046](#), [RFC 2047](#), [RFC 2183](#), et [RFC 2231](#).

Ce paquet peut être divisé entre trois composants majeurs, et un quatrième composant qui contrôle le comportement des trois autres.

Le composant central du paquet est un « modèle d'objet » qui représente les messages. Une application interagit avec le paquet, dans un premier temps, à travers l'interface de modèle d'objet définie dans le sous-module `message`. L'application peut utiliser cette API pour poser des questions à propos d'un mail existant, pour créer un nouvel e-mail, ou ajouter ou retirer des sous-composants d'e-mail qui utilisent la même interface de modèle d'objet. Suivant la nature des messages et leurs sous-composants MIME, le modèle d'objet d'e-mail est une structure arborescente d'objets qui fournit tout à l'API de `EmailMessage`.

Les deux autres composants majeurs de ce paquet sont l'analyseur (`parser`) et le générateur (`generator`). L'analyseur prend la version sérialisée d'un e-mail (un flux d'octets) et le convertit en une arborescence d'objets `EmailMessage`. Le générateur prend un objet `EmailMessage` et le retransforme en un flux d'octets sérialisé (l'analyseur et le générateur gèrent aussi des suites de caractères textuels, mais cette utilisation est déconseillée car il est très facile de finir avec des messages invalides d'une manière ou d'une autre).

Le composant de contrôle est le module `policy`. Chaque `EmailMessage`, chaque `generator` et chaque `parser` possède un objet associé `policy` qui contrôle son comportement. Habituellement une application n'a besoin de spécifier la politique que quand un `EmailMessage` est créé, soit en instanciant directement un `EmailMessage` pour créer un

nouvel e-mail, soit lors de l'analyse d'un flux entrant en utilisant un *parser*. Mais la politique peut être changée quand le message est sérialisé en utilisant un *generator*. Cela permet, par exemple, d'analyser un message e-mail générique du disque, puis de le sérialiser en utilisant une configuration SMTP standard quand on l'envoie vers un serveur d'e-mail.

Le paquet *email* fait son maximum pour cacher les détails des différentes RFCs de référence à l'application. Conceptuellement, l'application doit être capable de traiter l'e-mail comme une arborescence structurée de texte Unicode et de pièces jointes binaires, sans avoir à se préoccuper de leur représentation sérialisée. Dans la pratique, cependant, il est souvent nécessaire d'être conscient d'au moins quelques règles relatives aux messages MIME et à leur structure, en particulier les noms et natures des « types de contenus » et comment ils identifient les documents à plusieurs parties. Pour la plupart, cette connaissance devrait seulement être nécessaire pour des applications plus complexes, et même là, il devrait être question des structures de haut niveau et non des détails sur la manière dont elles sont représentées. Comme les types de contenus MIME sont couramment utilisés dans les logiciels internet modernes (et non uniquement les e-mails), les développeurs sont généralement familiers de ce concept.

La section suivante décrit les fonctionnalités du paquet *email*. Nous commençons avec le modèle d'objet *message*, qui est la principale interface qu'une application utilise, et continuons avec les composants *parser* et *generator*. Ensuite, nous couvrons les contrôles *policy*, qui complètent le traitement des principaux composants de la bibliothèque.

Les trois prochaines sections couvrent les exceptions que le paquet peut rencontrer et les imperfections (non-respect des RFCs) que le module *parser* peut détecter. Ensuite nous couvrons les sous-composants *headerregistry* et *contentmanager*, qui fournissent des outils pour faire des manipulations plus détaillées des en-têtes et du contenu, respectivement. Les deux composants contiennent des fonctionnalités adaptées pour traiter et produire des messages qui sortent de l'ordinaire, et elles documentent aussi leurs API pour pouvoir les étendre, ce qui ne manquera pas d'intéresser les applications avancées.

Ci-dessous se trouve un ensemble d'exemples d'utilisations des éléments fondamentaux des API couvertes dans les sections précédentes.

The foregoing represent the modern (unicode friendly) API of the email package. The remaining sections, starting with the *Message* class, cover the legacy *compat32* API that deals much more directly with the details of how email messages are represented. The *compat32* API does *not* hide the details of the RFCs from the application, but for applications that need to operate at that level, they can be useful tools. This documentation is also relevant for applications that are still using the *compat32* API for backward compatibility reasons.

Modifié dans la version 3.6 : Documents réorganisés et réécrits pour promouvoir la nouvelle API *EmailMessage/EmailPolicy*.

Contenus de la documentation du paquet *email* :

19.1.1 *email.message* : Representing an email message

Source code : <Lib/email/message.py>

Nouveau dans la version 3.6 : ¹

The central class in the *email* package is the *EmailMessage* class, imported from the *email.message* module. It is the base class for the *email* object model. *EmailMessage* provides the core functionality for setting and querying header fields, for accessing message bodies, and for creating or modifying structured messages.

An email message consists of *headers* and a *payload* (which is also referred to as the *content*). Headers are **RFC 5322** or **RFC 6532** style field names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

1. Originally added in 3.4 as a *provisional module*. Docs for legacy message class moved to *email.message.Message* : Representing an email message using the *compat32* API.

The conceptual model provided by an *EmailMessage* object is that of an ordered dictionary of headers coupled with a *payload* that represents the [RFC 5322](#) body of the message, which might be a list of sub-*EmailMessage* objects. In addition to the normal dictionary methods for accessing the header names and values, there are methods for accessing specialized information from the headers (for example the MIME content type), for operating on the payload, for generating a serialized version of the message, and for recursively walking over the object tree.

The *EmailMessage* dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. Unlike a real dict, there is an ordering to the keys, and there can be duplicate keys. Additional methods are provided for working with headers that have duplicate keys.

The *payload* is either a string or bytes object, in the case of simple message objects, or a list of *EmailMessage* objects, for MIME container documents such as *multipart/** and *message/rfc822* message objects.

class email.message.**EmailMessage** (*policy=default*)

If *policy* is specified use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *default* policy, which follows the rules of the email RFCs except for line endings (instead of the RFC mandated `\r\n`, it uses the Python standard `\n` line endings). For more information see the *policy* documentation.

as_string (*unixfrom=False, maxheaderlen=None, policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility with the base *Message* class *maxheaderlen* is accepted, but defaults to `None`, which means that by default the line length is controlled by the *max_line_length* of the policy. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *Generator*.

Flattening the message may trigger changes to the *EmailMessage* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.Generator* for a more flexible API for serializing messages. Note also that this method is restricted to producing messages serialized as « 7 bit clean » when *utf8* is `False`, which is the default.

Modifié dans la version 3.6 : the default behavior when *maxheaderlen* is not specified was changed from defaulting to 0 to defaulting to the value of *max_line_length* from the policy.

__str__ ()

Equivalent to `as_string(policy=self.policy.clone(utf8=True))`. Allows `str(msg)` to produce a string containing the serialized message in a readable format.

Modifié dans la version 3.4 : the method was changed to use `utf8=True`, thus producing an [RFC 6531](#)-like message representation, instead of being a direct alias for `as_string()`.

as_bytes (*unixfrom=False, policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *BytesGenerator*.

Flattening the message may trigger changes to the *EmailMessage* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.BytesGenerator* for a more flexible API for serializing messages.

__bytes__ ()

Equivalent to `as_bytes()`. Allows `bytes(msg)` to produce a bytes object containing the serialized message.

is_multipart ()

Return `True` if the message's payload is a list of sub-*EmailMessage* objects, otherwise return `False`.

When `is_multipart()` returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). Note that `is_multipart()` returning `True` does not necessarily mean that « `msg.get_content_maintype() == "multipart"` » will return the `True`. For example, `is_multipart` will return `True` when the `EmailMessage` is of type `message/rfc822`.

set_unixfrom (*unixfrom*)

Set the message's envelope header to *unixfrom*, which should be a string. (See `mboxMessage` for a brief description of this header.)

get_unixfrom ()

Return the message's envelope header. Defaults to `None` if the envelope header was never set.

The following methods implement the mapping-like interface for accessing the message's headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by `keys()`, but in an `EmailMessage` object, headers are always returned in the order they appeared in the original message, or in which they were added to the message later. Any header deleted and then re-added is always appended to the end of the header list.

These semantic differences are intentional and are biased toward convenience in the most common use cases.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

__len__ ()

Return the total number of headers, including duplicates.

__contains__ (*name*)

Return `true` if the message object has a field named *name*. Matching is done without regard to case and *name* does not include the trailing colon. Used for the `in` operator. For example :

```
if 'message-id' in myMessage:
    print('Message-ID:', myMessage['message-id'])
```

__getitem__ (*name*)

Return the value of the named header field. *name* does not include the colon field separator. If the header is missing, `None` is returned; a `KeyError` is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the `get_all()` method to get the values of all the extant headers named *name*.

Using the standard (non-compat32) policies, the returned value is an instance of a subclass of `email.headerregistry.BaseHeader`.

__setitem__ (*name*, *val*)

Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing headers.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g. :

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

If the policy defines certain headers to be unique (as the standard policies do), this method may raise a `ValueError` when an attempt is made to assign a value to such a header when one already exists. This behavior is intentional for consistency's sake, but do not depend on it as we may choose to make such assignments do an automatic deletion of the existing header in the future.

__delitem__ (*name*)

Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys ()

Return a list of all the message's header field names.

values ()

Return a list of all the message's field values.

items()

Return a list of 2-tuples containing all the message's field headers and values.

get (*name*, *failobj*=None)

Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (*failobj* defaults to None).

Here are some additional useful header related methods :

get_all (*name*, *failobj*=None)

Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to None).

add_header (*_name*, *_value*, ***_params*)

Extended header setting. This method is similar to `__setitem__()` except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is None, in which case only the key will be added.

If the value contains non-ASCII characters, the charset and language may be explicitly controlled by specifying the value as a three tuple in the format (CHARSET, LANGUAGE, VALUE), where CHARSET is a string naming the charset to be used to encode the value, LANGUAGE can usually be set to None or the empty string (see [RFC 2231](#) for other possibilities), and VALUE is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in [RFC 2231](#) format using a CHARSET of `utf-8` and a LANGUAGE of None.

Voici un exemple :

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example of the extended interface with non-ASCII characters :

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header (*_name*, *_value*)

Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case of the original header. If no matching header is found, raise a `KeyError`.

get_content_type ()

Return the message's content type, coerced to lower case of the form *maintype/subtype*. If there is no *Content-Type* header in the message return the value returned by `get_default_type()`. If the *Content-Type* header is invalid, return `text/plain`.

(According to [RFC 2045](#), messages always have a default type, `get_content_type()` will always return a value. [RFC 2045](#) defines a message's default type to be `text/plain` unless it appears inside a *multipart/digest* container, in which case it would be `message/rfc822`. If the *Content-Type* header has an invalid type specification, [RFC 2045](#) mandates that the default type be `text/plain`.)

get_content_maintype ()

Return the message's main content type. This is the *maintype* part of the string returned by `get_content_type()`.

get_content_subtype ()

Return the message's sub-content type. This is the *subtype* part of the string returned by `get_content_type()`.

get_default_type ()

Return the default content type. Most messages have a default content type of `text/plain`, except for

messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type (*ctype*)

Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header, so it only affects the return value of the *get_content_type* methods when no *Content-Type* header is present in the message.

set_param (*param*, *value*, *header*='Content-Type', *quote*=True, *charset*=None, *language*='', *replace*=False)

Set a parameter in the *Content-Type* header. If the parameter already exists in the header, replace its value with *value*. When *header* is *Content-Type* (the default) and the header does not yet exist in the message, add it, set its value to *text/plain*, and append the new parameter value. Optional *header* specifies an alternative header to *Content-Type*.

If the value contains non-ASCII characters, the *charset* and *language* may be explicitly specified using the optional *charset* and *language* parameters. Optional *language* specifies the **RFC 2231** language, defaulting to the empty string. Both *charset* and *language* should be strings. The default is to use the *utf8* *charset* and *None* for the *language*.

If *replace* is *False* (the default) the header is moved to the end of the list of headers. If *replace* is *True*, the header will be updated in place.

Use of the *quote* parameter with *EmailMessage* objects is deprecated.

Note that existing parameter values of headers may be accessed through the *params* attribute of the header value (for example, `msg['Content-Type'].params['charset']`).

Modifié dans la version 3.4 : *replace* keyword was added.

del_param (*param*, *header*='content-type', *quote*=True)

Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. Optional *header* specifies an alternative to *Content-Type*.

Use of the *quote* parameter with *EmailMessage* objects is deprecated.

get_filename (*failobj*=None)

Return the value of the *filename* parameter of the *Content-Disposition* header of the message. If the header does not have a *filename* parameter, this method falls back to looking for the *name* parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per `email.utils.unquote()`.

get_boundary (*failobj*=None)

Return the value of the *boundary* parameter of the *Content-Type* header of the message, or *failobj* if either the header is missing, or has no *boundary* parameter. The returned string will always be unquoted as per `email.utils.unquote()`.

set_boundary (*boundary*)

Set the *boundary* parameter of the *Content-Type* header to *boundary*. `set_boundary()` will always quote *boundary* if necessary. A *HeaderParseError* is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different from deleting the old *Content-Type* header and adding a new one with the new *boundary* via `add_header()`, because `set_boundary()` preserves the order of the *Content-Type* header in the list of headers.

get_content_charset (*failobj*=None)

Return the *charset* parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no *charset* parameter, *failobj* is returned.

get_charsets (*failobj*=None)

Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the *charset* parameter in the *Content-Type* header for the represented subpart. If the subpart has no *Content-Type* header, no *charset* parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

is_attachment()

Return True if there is a *Content-Disposition* header and its (case insensitive) value is attachment, False otherwise.

Modifié dans la version 3.4.2 : `is_attachment` is now a method instead of a property, for consistency with `is_multipart()`.

get_content_disposition()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or None. The possible values for this method are *inline*, *attachment* or None if the message follows **RFC 2183**.

Nouveau dans la version 3.5.

The following methods relate to interrogating and manipulating the content (payload) of the message.

walk()

The `walk()` method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use `walk()` as the iterator in a `for` loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure :

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where `is_multipart()` returns True, even though `msg.get_content_maintype() == 'multipart'` may return False. We can see this in our example by making use of the `_structure` debug helper function :

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
  text/plain
  message/delivery-status
    text/plain
    text/plain
  message/rfc822
    text/plain
```

Here the message parts are not multipart, but they do contain subparts. `is_multipart()` returns True and `walk` descends into the subparts.

get_body(preferencelist=('related', 'html', 'plain'))

Return the MIME part that is the best candidate to be the « body » of the message.

`preferencelist` must be a sequence of strings from the set `related`, `html`, and `plain`, and indicates the order of preference for the content type of the part returned.

Start looking for candidate matches with the object on which the `get_body` method is called.

If `related` is not included in *preferencelist*, consider the root part (or subpart of the root part) of any related encountered as a candidate if the (sub-)part matches a preference.

When encountering a multipart/related, check the `start` parameter and if a part with a matching *Content-ID* is found, consider only it when looking for candidate matches. Otherwise consider only the first (default root) part of the multipart/related.

If a part has a *Content-Disposition* header, only consider the part a candidate match if the value of the header is `inline`.

If none of the candidates matches any of the preferences in *preferencelist*, return `None`.

Notes : (1) For most applications the only *preferencelist* combinations that really make sense are `('plain',)`, `('html', 'plain')`, and the default `('related', 'html', 'plain')`. (2) Because matching starts with the object on which `get_body` is called, calling `get_body` on a multipart/related will return the object itself unless *preferencelist* has a non-default value. (3) Messages (or message parts) that do not specify a *Content-Type* or whose *Content-Type* header is invalid will be treated as if they are of type `text/plain`, which may occasionally cause `get_body` to return unexpected results.

iter_attachments()

Return an iterator over all of the immediate sub-parts of the message that are not candidate « body » parts. That is, skip the first occurrence of each of `text/plain`, `text/html`, `multipart/related`, or `multipart/alternative` (unless they are explicitly marked as attachments via *Content-Disposition: attachment*), and return all remaining parts. When applied directly to a multipart/related, return an iterator over the all the related parts except the root part (ie : the part pointed to by the `start` parameter, or the first part if there is no `start` parameter or the `start` parameter doesn't match the *Content-ID* of any of the parts). When applied directly to a multipart/alternative or a non-multipart, return an empty iterator.

iter_parts()

Return an iterator over all of the immediate sub-parts of the message, which will be empty for a non-multipart. (See also `walk()`.)

get_content(*args, content_manager=None, **kw)

Call the `get_content()` method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the *content_manager* specified by the current *policy*.

set_content(*args, content_manager=None, **kw)

Call the `set_content()` method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the *content_manager* specified by the current *policy*.

make_related(boundary=None)

Convert a non-multipart message into a multipart/related message, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_alternative(boundary=None)

Convert a non-multipart or a multipart/related into a multipart/alternative, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_mixed(boundary=None)

Convert a non-multipart, a multipart/related, or a multipart-alternative into a multipart/mixed, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

add_related(*args, content_manager=None, **kw)

If the message is a multipart/related, create a new message object, pass all of the arguments to its

`set_content()` method, and `attach()` it to the multipart. If the message is a non-multipart, call `make_related()` and then proceed as above. If the message is any other type of multipart, raise a `TypeError`. If `content_manager` is not specified, use the `content_manager` specified by the current `policy`. If the added part has no `Content-Disposition` header, add one with the value `inline`.

add_alternative (*args, content_manager=None, **kw)

If the message is a multipart/alternative, create a new message object, pass all of the arguments to its `set_content()` method, and `attach()` it to the multipart. If the message is a non-multipart or multipart/related, call `make_alternative()` and then proceed as above. If the message is any other type of multipart, raise a `TypeError`. If `content_manager` is not specified, use the `content_manager` specified by the current `policy`.

add_attachment (*args, content_manager=None, **kw)

If the message is a multipart/mixed, create a new message object, pass all of the arguments to its `set_content()` method, and `attach()` it to the multipart. If the message is a non-multipart, multipart/related, or multipart/alternative, call `make_mixed()` and then proceed as above. If `content_manager` is not specified, use the `content_manager` specified by the current `policy`. If the added part has no `Content-Disposition` header, add one with the value `attachment`. This method can be used both for explicit attachments (`Content-Disposition: attachment`) and inline attachments (`Content-Disposition: inline`), by passing appropriate options to the `content_manager`.

clear ()

Remove the payload and all of the headers.

clear_content ()

Remove the payload and all of the `Content-` headers, leaving all other headers intact and in their original order.

`EmailMessage` objects have the following instance attributes :

preamble

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The `preamble` attribute contains this leading extra-armor text for MIME documents. When the `Parser` discovers some text after the headers but before the first boundary string, it assigns this text to the message's `preamble` attribute. When the `Generator` is writing out the plain text representation of a MIME message, and it finds the message has a `preamble` attribute, it will write this text in the area between the headers and the first boundary. See `email.parser` and `email.generator` for details.

Note that if the message object has no preamble, the `preamble` attribute will be `None`.

epilogue

The `epilogue` attribute acts the same way as the `preamble` attribute, except that it contains text that appears between the last boundary and the end of the message. As with the `preamble`, if there is no epilog text this attribute will be `None`.

defects

The `defects` attribute contains a list of all the problems found when parsing this message. See `email.errors` for a detailed description of the possible parsing defects.

class `email.message.MIMEPart` (policy=default)

This class represents a subpart of a MIME message. It is identical to `EmailMessage`, except that no `MIME-Version` headers are added when `set_content()` is called, since sub-parts do not need their own `MIME-Version` headers.

Notes

19.1.2 `email.parser` : Analyser des e-mails

Code source : [Lib/email/parser.py](#)

Les instances de messages peuvent être créées de deux façons : elles peuvent être créées de toutes pièces en créant un objet `EmailMessage`, en ajoutant des en-têtes en utilisant l'interface de dictionnaire, et en ajoutant un ou plusieurs corps de message en utilisant `set_content()` et les méthodes associées, ou ils peuvent être créés en analysant une représentation sérialisée de l'e-mail.

The `email` package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a bytes, string or file object, and the parser will return to you the root `EmailMessage` instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return `True` from its `is_multipart()` method, and the subparts can be accessed via the payload manipulation methods, such as `get_body()`, `iter_parts()`, and `walk()`.

There are actually two parser interfaces available for use, the `Parser` API and the incremental `FeedParser` API. The `Parser` API is most useful if you have the entire text of the message in memory, or if the entire message lives in a file on the file system. `FeedParser` is more appropriate when you are reading the message from a stream which might block waiting for more input (such as reading an email message from a socket). The `FeedParser` can consume and parse the message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. All of the logic that connects the `email` package's bundled parser and the `EmailMessage` class is embodied in the `policy` class, so a custom parser can create message object trees any way it finds necessary by implementing custom versions of the appropriate `policy` methods.

API `FeedParser`

The `BytesFeedParser`, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (such as a socket). The `BytesFeedParser` can of course be used to parse an email message fully contained in a *bytes-like object*, string, or file, but the `BytesParser` API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The `BytesFeedParser`'s API is simple ; you create an instance, feed it a bunch of bytes until there's no more to feed it, then close the parser to retrieve the root message object. The `BytesFeedParser` is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's `defects` attribute with a list of any problems it found in a message. See the `email.errors` module for the list of defects that it can find.

Voici l'API pour `BytesFeedParser` :

class `email.parser.BytesFeedParser` (`_factory=None`, *, `policy=policy.compat32`)

Create a `BytesFeedParser` instance. Optional `_factory` is a no-argument callable; if not specified use the `message_factory` from the `policy`. Call `_factory` whenever a new message object is needed.

If `policy` is specified use the rules it specifies to update the representation of the message. If `policy` is not set, use the `compat32` policy, which maintains backward compatibility with the Python 3.2 version of the email package and provides `Message` as the default factory. All other policies provide `EmailMessage` as the default `_factory`. For more information on what else `policy` controls, see the `policy` documentation.

Note : The **policy** keyword should always be specified ; The default will change to `email.policy.default` in a future version of Python.

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : Added the *policy* keyword.

Modifié dans la version 3.6 : *_factory* defaults to the policy *message_factory*.

feed (*data*)

Feed the parser some more data. *data* should be a *bytes-like object* containing one or more lines. The lines can be partial and the parser will stitch such partial lines together properly. The lines can have any of the three common line endings : carriage return, newline, or carriage return and newline (they can even be mixed).

close ()

Complete the parsing of all previously fed data and return the root message object. It is undefined what happens if *feed()* is called after this method has been called.

class `email.parser.FeedParser` (*_factory=None*, *, *policy=policy.compat32*)

Works like *BytesFeedParser* except that the input to the *feed()* method must be a string. This is of limited utility, since the only way for such a message to be valid is for it to contain only ASCII text or, if *utf8* is *True*, no binary attachments.

Modifié dans la version 3.3 : Added the *policy* keyword.

API de Parser

The *BytesParser* class, imported from the *email.parser* module, provides an API that can be used to parse a message when the complete contents of the message are available in a *bytes-like object* or file. The *email.parser* module also provides *Parser* for parsing strings, and header-only parsers, *BytesHeaderParser* and *HeaderParser*, which can be used if you're only interested in the headers of the message. *BytesHeaderParser* and *HeaderParser* can be much faster in these situations, since they do not attempt to parse the message body, instead setting the payload to the raw body.

class `email.parser.BytesParser` (*_class=None*, *, *policy=policy.compat32*)

Create a *BytesParser* instance. The *_class* and *policy* arguments have the same meaning and semantics as the *_factory* and *policy* arguments of *BytesFeedParser*.

Note : **The policy keyword should always be specified** ; The default will change to *email.policy.default* in a future version of Python.

Modifié dans la version 3.3 : Removed the *strict* argument that was deprecated in 2.4. Added the *policy* keyword.

Modifié dans la version 3.6 : *_class* defaults to the policy *message_factory*.

parse (*fp*, *headersonly=False*)

Read all the data from the binary file-like object *fp*, parse the resulting bytes, and return the message object. *fp* must support both the *readline()* and the *read()* methods.

The bytes contained in *fp* must be formatted as a block of **RFC 5322** (or, if *utf8* is *True*, **RFC 6532**) style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts, including subparts with a *Content-Transfer-Encoding* of 8bit).

Optional *headersonly* is a flag specifying whether to stop parsing after reading the headers or not. The default is *False*, meaning it parses the entire contents of the file.

parsebytes (*bytes*, *headersonly=False*)

Similar to the *parse()* method, except it takes a *bytes-like object* instead of a file-like object. Calling this method on a *bytes-like object* is equivalent to wrapping *bytes* in a *BytesIO* instance first and calling *parse()*.

Optional *headersonly* is as with the *parse()* method.

Nouveau dans la version 3.2.

class `email.parser.BytesHeaderParser` (*_class=None*, *, *policy=policy.compat32*)

Exactly like *BytesParser*, except that *headersonly* defaults to *True*.

Nouveau dans la version 3.3.

class email.parser.Parser(_class=None, *, policy=policy.compat32)

This class is parallel to [BytesParser](#), but handles string input.

Modifié dans la version 3.3 : Removed the *strict* argument. Added the *policy* keyword.

Modifié dans la version 3.6 : *_class* defaults to the policy *message_factory*.

parse(fp, headersonly=False)

Read all the data from the text-mode file-like object *fp*, parse the resulting text, and return the root message object. *fp* must support both the [readline\(\)](#) and the [read\(\)](#) methods on file-like objects.

Other than the text mode requirement, this method operates like [BytesParser.parse\(\)](#).

parsestr(text, headersonly=False)

Similar to the [parse\(\)](#) method, except it takes a string object instead of a file-like object. Calling this method on a string is equivalent to wrapping *text* in a [StringIO](#) instance first and calling [parse\(\)](#).

Optional *headersonly* is as with the [parse\(\)](#) method.

class email.parser.HeaderParser(_class=None, *, policy=policy.compat32)

Exactly like [Parser](#), except that *headersonly* defaults to True.

Since creating a message object structure from a string or a file object is such a common task, four functions are provided as a convenience. They are available in the top-level [email](#) package namespace.

email.message_from_bytes(s, _class=None, *, policy=policy.compat32)

Return a message object structure from a *bytes-like object*. This is equivalent to [BytesParser\(\)](#).
[parsebytes\(s\)](#). Optional *_class* and *policy* are interpreted as with the [BytesParser](#) class constructor.

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : Removed the *strict* argument. Added the *policy* keyword.

email.message_from_binary_file(fp, _class=None, *, policy=policy.compat32)

Return a message object structure tree from an open binary *file object*. This is equivalent to [BytesParser\(\)](#).
[parse\(fp\)](#). *_class* and *policy* are interpreted as with the [BytesParser](#) class constructor.

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : Removed the *strict* argument. Added the *policy* keyword.

email.message_from_string(s, _class=None, *, policy=policy.compat32)

Return a message object structure from a string. This is equivalent to [Parser\(\)](#).
[parsestr\(s\)](#). *_class* and *policy* are interpreted as with the [Parser](#) class constructor.

Modifié dans la version 3.3 : Removed the *strict* argument. Added the *policy* keyword.

email.message_from_file(fp, _class=None, *, policy=policy.compat32)

Return a message object structure tree from an open *file object*. This is equivalent to [Parser\(\)](#).
[parse\(fp\)](#). *_class* and *policy* are interpreted as with the [Parser](#) class constructor.

Modifié dans la version 3.3 : Removed the *strict* argument. Added the *policy* keyword.

Modifié dans la version 3.6 : *_class* defaults to the policy *message_factory*.

Here's an example of how you might use [message_from_bytes\(\)](#) at an interactive Python prompt :

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

Notes complémentaires

Voici des remarques sur la sémantique d'analyse :

- Most non-*multipart* type messages are parsed as a single message object with a string payload. These objects will return `False` for `is_multipart()`, and `iter_parts()` will yield an empty list.
- All *multipart* type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for `is_multipart()`, and `iter_parts()` will yield a list of subparts.
- Most messages with a content type of *message/** (such as *message/delivery-status* and *message/rfc822*) will also be parsed as container object containing a list payload of length 1. Their `is_multipart()` method will return `True`. The single element yielded by `iter_parts()` will be a sub-message object.
- Some non-standards-compliant messages may not be internally consistent about their *multipart*-edness. Such messages may have a *Content-Type* header of type *multipart*, but their `is_multipart()` method may return `False`. If such messages were parsed with the *FeedParser*, they will have an instance of the `MultipartInvariantViolationDefect` class in their `defects` attribute list. See `email.errors` for details.

19.1.3 `email.generator`: Generating MIME documents

Source code : [Lib/email/generator.py](#)

One of the most common tasks is to generate the flat (serialized) version of the email message represented by a message object structure. You will need to do this if you want to send your message via `smtplib.SMTP.sendmail()` or the `nntplib` module, or print the message on the console. Taking a message object structure and producing a serialized representation is the job of the generator classes.

As with the `email.parser` module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the bytes-oriented parsing and generation operations are inverses, assuming the same non-transforming *policy* is used for both. That is, parsing the serialized byte stream via the `BytesParser` class and then regenerating the serialized byte stream using `BytesGenerator` should produce output identical to the input¹. (On the other hand, using the generator on an `EmailMessage` constructed by program may result in changes to the `EmailMessage` object as defaults are filled in.)

The `Generator` class can be used to flatten a message into a text (as opposed to binary) serialized representation, but since Unicode cannot represent binary data directly, the message is of necessity transformed into something that contains only ASCII characters, using the standard email RFC Content Transfer Encoding techniques for encoding email messages for transport over channels that are not « 8 bit clean ».

class `email.generator.BytesGenerator` (*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *, *policy=None*)

Return a `BytesGenerator` object that will write any message provided to the `flatten()` method, or any surrogateescape encoded text provided to the `write()` method, to the *file-like object* *outfp*. *outfp* must support a `write` method that accepts binary data.

If optional *mangle_from_* is `True`, put a > character in front of any line in the body that starts with the exact string "From ", that is From followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is `True` for the `compat32` policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in unix mbox format (see `mailbox` and **WHY THE CONTENT-LENGTH FORMAT IS BAD**).

1. This statement assumes that you use the appropriate setting for `unixfrom`, and that there are no *policy* settings calling for automatic adjustments (for example, `refold_source` must be `none`, which is *not* the default). It is also not 100% true, since if the message does not conform to the RFC standards occasionally information about the exact original text is lost during parsing error recovery. It is a goal to fix these latter edge cases when possible.

If *maxheaderlen* is not *None*, refold any header lines that are longer than *maxheaderlen*, or if 0, do not rewrap any headers. If *manheaderlen* is *None* (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is *None* (the default), use the policy associated with the *Message* or *EmailMessage* object passed to *flatten* to control the message generation. See *email.policy* for details on what *policy* controls.

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : Added the *policy* keyword.

Modifié dans la version 3.6 : The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten (*msg*, *unixfrom=False*, *linesep=None*)

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the *BytesGenerator* instance was created.

If the *policy* option *cte_type* is 8bit (the default), copy any headers in the original parsed message that have not been modified to the output with any bytes with the high bit set reproduced as in the original, and preserve the non-ASCII *Content-Transfer-Encoding* of any body parts that have them. If *cte_type* is 7bit, convert the bytes with the high bit set as needed using an ASCII-compatible *Content-Transfer-Encoding*. That is, transform parts with non-ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) to an ASCII compatible *Content-Transfer-Encoding*, and encode RFC-invalid non-ASCII bytes in headers using the MIME unknown-8bit character set, thus rendering them RFC-compliant.

If *unixfrom* is *True*, print the envelope header delimiter used by the Unix mailbox format (see *mailbox*) before the first of the [RFC 5322](#) headers of the root message object. If the root object has no envelope header, craft a standard one. The default is *False*. Note that for subparts, no envelope header is ever printed.

If *linesep* is not *None*, use it as the separator character between all the lines of the flattened message. If *linesep* is *None* (the default), use the value specified in the *policy*.

clone (*fp*)

Return an independent clone of this *BytesGenerator* instance with the exact same option settings, and *fp* as the new *outfp*.

write (*s*)

Encode *s* using the ASCII codec and the *surrogateescape* error handler, and pass it to the *write* method of the *outfp* passed to the *BytesGenerator*'s constructor.

As a convenience, *EmailMessage* provides the methods *as_bytes()* and *bytes(aMessage)* (a.k.a. *__bytes__()*), which simplify the generation of a serialized binary representation of a message object. For more detail, see *email.message*.

Because strings cannot represent binary data, the *Generator* class must convert any binary data in any message it flattens to an ASCII compatible format, by converting them to an ASCII compatible *Content-Transfer-Encoding*. Using the terminology of the email RFCs, you can think of this as *Generator* serializing to an I/O stream that is not « 8 bit clean ». In other words, most applications will want to be using *BytesGenerator*, and not *Generator*.

class email.generator.**Generator** (*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *, *policy=None*)

Return a *Generator* object that will write any message provided to the *flatten()* method, or any text provided to the *write()* method, to the *file-like object* *outfp*. *outfp* must support a *write* method that accepts string data.

If optional *mangle_from_* is *True*, put a > character in front of any line in the body that starts with the exact string "From ", that is From followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is *True* for the *compat32* policy and *False* for all others). *mangle_from_* is intended for use when messages are stored in unix mbox format (see *mailbox* and [WHY THE CONTENT-LENGTH FORMAT IS BAD](#)).

If *maxheaderlen* is not *None*, refold any header lines that are longer than *maxheaderlen*, or if 0, do not rewrap any headers. If *manheaderlen* is *None* (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the `Message` or `EmailMessage` object passed to `flatten` to control the message generation. See `email.policy` for details on what *policy* controls.

Modifié dans la version 3.3 : Added the *policy* keyword.

Modifié dans la version 3.6 : The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten (*msg*, *unixfrom=False*, *linesep=None*)

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the `Generator` instance was created.

If the *policy* option *cte_type* is `8bit`, generate the message as if the option were set to `7bit`. (This is required because strings cannot represent non-ASCII bytes.) Convert any bytes with the high bit set as needed using an ASCII-compatible `Content-Transfer-Encoding`. That is, transform parts with non-ASCII `Cotnent-Transfer-Encoding` (`Content-Transfer-Encoding: 8bit`) to an ASCII compatible `Content-Transfer-Encoding`, and encode RFC-invalid non-ASCII bytes in headers using the MIME unknown-8bit character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see `mailbox`) before the first of the [RFC 5322](#) headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

Modifié dans la version 3.2 : Added support for re-encoding `8bit` message bodies, and the *linesep* argument.

clone (*fp*)

Return an independent clone of this `Generator` instance with the exact same options, and *fp* as the new *outfp*.

write (*s*)

Write *s* to the `write` method of the *outfp* passed to the `Generator`'s constructor. This provides just enough file-like API for `Generator` instances to be used in the `print()` function.

As a convenience, `EmailMessage` provides the methods `as_string()` and `str(aMessage)` (a.k.a. `__str__()`), which simplify the generation of a formatted string representation of a message object. For more detail, see `email.message`.

The `email.generator` module also provides a derived class, `DecodedGenerator`, which is like the `Generator` base class, except that non-*text* parts are not serialized, but are instead represented in the output stream by a string derived from a template filled in with information about the part.

class `email.generator.DecodedGenerator` (*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, *, *policy=None*)

Act like `Generator`, except that for any subpart of the message passed to `Generator.flatten()`, if the subpart is of main type *text*, print the decoded payload of the subpart, and if the main type is not *text*, instead of printing it fill in the string *fmt* using information from the part and print the resulting filled-in string.

To fill in *fmt*, execute `fmt % part_info`, where `part_info` is a dictionary composed of the following keys and values :

- *type* — Full MIME type of the non-*text* part
- *maintype* — Main MIME type of the non-*text* part
- *subtype* — Sub-MIME type of the non-*text* part
- *filename* — Filename of the non-*text* part
- *description* — Description associated with the non-*text* part
- *encoding* — Content transfer encoding of the non-*text* part

If *fmt* is `None`, use the following default *fmt* :

« [Non-text (%(type)s) part of message omitted, filename %(filename)s] »

Optional *_mangle_from_* and *maxheaderlen* are as with the `Generator` base class.

Notes

19.1.4 `email.policy` : Policy Objects

Nouveau dans la version 3.3.

Source code : [Lib/email/policy.py](#)

The `email` package's prime focus is the handling of email messages as described by the various email and MIME RFCs. However, the general format of email messages (a block of header fields each consisting of a name followed by a colon followed by a value, the whole block followed by a blank line and an arbitrary "body"), is a format that has found utility outside of the realm of email. Some of these uses conform fairly closely to the main email RFCs, some do not. Even when working with email, there are times when it is desirable to break strict compliance with the RFCs, such as generating emails that interoperate with email servers that do not themselves follow the standards, or that implement extensions you want to use in ways that violate the standards.

Policy objects give the email package the flexibility to handle all these disparate use cases.

A `Policy` object encapsulates a set of attributes and methods that control the behavior of various components of the email package during use. `Policy` instances can be passed to various classes and methods in the email package to alter the default behavior. The settable values and their defaults are described below.

There is a default policy used by all classes in the email package. For all of the `parser` classes and the related convenience functions, and for the `Message` class, this is the `Compat32` policy, via its corresponding pre-defined instance `compat32`. This policy provides for complete backward compatibility (in some cases, including bug compatibility) with the pre-Python3.3 version of the email package.

This default value for the `policy` keyword to `EmailMessage` is the `EmailPolicy` policy, via its pre-defined instance `default`.

When a `Message` or `EmailMessage` object is created, it acquires a policy. If the message is created by a `parser`, a policy passed to the parser will be the policy used by the message it creates. If the message is created by the program, then the policy can be specified when it is created. When a message is passed to a `generator`, the generator uses the policy from the message by default, but you can also pass a specific policy to the generator that will override the one stored on the message object.

The default value for the `policy` keyword for the `email.parser` classes and the parser convenience functions **will be changing** in a future version of Python. Therefore you should **always specify explicitly which policy you want to use** when calling any of the classes and functions described in the `parser` module.

The first part of this documentation covers the features of `Policy`, an *abstract base class* that defines the features that are common to all policy objects, including `compat32`. This includes certain hook methods that are called internally by the email package, which a custom policy could override to obtain different behavior. The second part describes the concrete classes `EmailPolicy` and `Compat32`, which implement the hooks that provide the standard behavior and the backward compatible behavior and features, respectively.

`Policy` instances are immutable, but they can be cloned, accepting the same keyword arguments as the class constructor and returning a new `Policy` instance that is a copy of the original but with the specified attributes values changed.

As an example, the following code could be used to read an email message from a file on disk and pass it to the system `sendmail` program on a Unix system :

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     msg = message_from_binary_file(f, policy=policy.default)
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Here we are telling *BytesGenerator* to use the RFC correct line separator characters when creating the binary string to feed into *sendmail*'s *stdin*, where the default policy would use `\n` line separators.

Some email package methods accept a *policy* keyword argument, allowing the policy to be overridden for that method. For example, the following code uses the *as_bytes()* method of the *msg* object from the previous example and writes the message to a file using the native line separators for the platform on which it is running :

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Policy objects can also be combined using the addition operator, producing a policy object whose settings are a combination of the non-default values of the summed objects :

```
>>> compat SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict SMTP = compat SMTP + compat_strict
```

This operation is not commutative ; that is, the order in which the objects are added matters. To illustrate :

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.Policy(**kw)

This is the *abstract base class* for all policy classes. It provides default implementations for a couple of trivial methods, as well as the implementation of the immutability property, the *clone()* method, and the constructor semantics.

The constructor of a policy class can be passed various keyword arguments. The arguments that may be specified are any non-method properties on this class, plus any additional non-method properties on the concrete class. A value specified in the constructor will override the default value for the corresponding attribute.

This class defines the following properties, and thus values for the following may be passed in the constructor of any policy class :

max_line_length

The maximum length of any line in the serialized output, not counting the end of line character(s). Default is 78, per **RFC 5322**. A value of 0 or *None* indicates that no line wrapping should be done at all.

linesep

The string to be used to terminate lines in serialized output. The default is `\n` because that's the internal end-of-line discipline used by Python, though `\r\n` is required by the RFCs.

cte_type

Controls the type of Content Transfer Encodings that may be or are required to be used. The possible values are :

7bit	all data must be « 7 bit clean » (ASCII-only). This means that where necessary data will be encoded using either quoted-printable or base64 encoding.
8bit	data is not constrained to be 7 bit clean. Data in headers is still required to be ASCII-only and so will be encoded (see <code>fold_binary()</code> and <code>utf8</code> below for exceptions), but body parts may use the 8bit CTE.

A `cte_type` value of 8bit only works with `BytesGenerator`, not `Generator`, because strings cannot contain binary data. If a `Generator` is operating under a policy that specifies `cte_type=8bit`, it will act as if `cte_type` is 7bit.

raise_on_defect

If `True`, any defects encountered will be raised as errors. If `False` (the default), defects will be passed to the `register_defect()` method.

mangle_from_

If `True`, lines starting with « *From* » in the body are escaped by putting a > in front of them. This parameter is used when the message is being serialized by a generator. Default : `False`.

Nouveau dans la version 3.5 : The `mangle_from_` parameter.

message_factory

A factory function for constructing a new empty message object. Used by the parser when building messages. Defaults to `None`, in which case `Message` is used.

Nouveau dans la version 3.6.

The following `Policy` method is intended to be called by code using the email library to create policy instances with custom settings :

clone (***kw*)

Return a new `Policy` instance whose attributes have the same values as the current instance, except where those attributes are given new values by the keyword arguments.

The remaining `Policy` methods are called by the email package code, and are not intended to be called by an application using the email package. A custom policy must implement all of these methods.

handle_defect (*obj, defect*)

Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of `Defect`.

The default implementation checks the `raise_on_defect` flag. If it is `True`, *defect* is raised as an exception. If it is `False` (the default), *obj* and *defect* are passed to `register_defect()`.

register_defect (*obj, defect*)

Register a *defect* on *obj*. In the email package, *defect* will always be a subclass of `Defect`.

The default implementation calls the `append` method of the `defects` attribute of *obj*. When the email package calls `handle_defect`, *obj* will normally have a `defects` attribute that has an `append` method. Custom object types used with the email package (for example, custom `Message` objects) should also provide such an attribute, otherwise defects in parsed messages will raise unexpected errors.

header_max_count (*name*)

Return the maximum allowed number of headers named *name*.

Called when a header is added to an `EmailMessage` or `Message` object. If the returned value is not 0 or `None`, and there are already a number of headers with the name *name* greater than or equal to the value returned, a `ValueError` is raised.

Because the default behavior of `Message.__setitem__` is to append the value to the list of headers, it is easy to create duplicate headers without realizing it. This method allows certain headers to be limited in the number of instances of that header that may be added to a `Message` programmatically. (The limit is not observed by the parser, which will faithfully produce as many headers as exist in the message being parsed.)

The default implementation returns `None` for all header names.

header_source_parse (*sourcelines*)

The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace

in the source is preserved. The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the parsed header.

If an implementation wishes to retain compatibility with the existing email package policies, *name* should be the case preserved name (all characters up to the “:” separator), while *value* should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.

sourcelines may contain surrogateescaped binary data.

There is no default implementation

header_store_parse (*name, value*)

The email package calls this method with the *name* and *value* provided by the application program when the application program is modifying a `Message` programmatically (as opposed to a `Message` created by a parser). The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the header.

If an implementation wishes to retain compatibility with the existing email package policies, the *name* and *value* should be strings or string subclasses that do not change the content of the passed in arguments.

There is no default implementation

header_fetch_parse (*name, value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` when that header is requested by the application program, and whatever the method returns is what is passed back to the application as the value of the header being retrieved. Note that there may be more than one header with the same name stored in the `Message`; the method is passed the specific name and value of the header destined to be returned to the application.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the value returned by the method.

There is no default implementation

fold (*name, value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` for a given header. The method should return a string that represents that header « folded » correctly (according to the policy settings) by composing the *name* with the *value* and inserting `linesep` characters at the appropriate places. See [RFC 5322](#) for a discussion of the rules for folding email headers.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the string returned by the method.

fold_binary (*name, value*)

The same as `fold()`, except that the returned value should be a bytes object rather than a string.

value may contain surrogateescaped binary data. These could be converted back into binary data in the returned bytes object.

class `email.policy.EmailPolicy` (***kw*)

This concrete *Policy* provides behavior that is intended to be fully compliant with the current email RFCs. These include (but are not limited to) [RFC 5322](#), [RFC 2047](#), and the current MIME RFCs.

This policy adds new header parsing and folding algorithms. Instead of simple strings, headers are `str` subclasses with attributes that depend on the type of the field. The parsing and folding algorithm fully implement [RFC 2047](#) and [RFC 5322](#).

The default value for the *message_factory* attribute is `EmailMessage`.

In addition to the settable attributes listed above that apply to all policies, this policy adds the following additional attributes :

Nouveau dans la version 3.6 :¹

utf8

If `False`, follow [RFC 5322](#), supporting non-ASCII characters in headers by encoding them as « encoded words ». If `True`, follow [RFC 6532](#) and use `utf-8` encoding for headers. Messages formatted in this way may be passed to SMTP servers that support the SMTPUTF8 extension ([RFC 6531](#)).

1. Originally added in 3.3 as a *provisional feature*.

refold_source

If the value for a header in the `Message` object originated from a `parser` (as opposed to being set by a program), this attribute indicates whether or not a generator should refold that value when transforming the message back into serialized form. The possible values are :

<code>none</code>	all source values use original folding
<code>long</code>	source values that have any line that is longer than <code>max_line_length</code> will be refolded
<code>all</code>	all values are refolded.

The default is `long`.

header_factory

A callable that takes two arguments, `name` and `value`, where `name` is a header field name and `value` is an unfolded header field value, and returns a string subclass that represents that header. A default `header_factory` (see `headerregistry`) is provided that supports custom parsing for the various address and date **RFC 5322** header field types, and the major MIME header field types. Support for additional custom parsing will be added in the future.

content_manager

An object with at least two methods : `get_content` and `set_content`. When the `get_content()` or `set_content()` method of an `EmailMessage` object is called, it calls the corresponding method of this object, passing it the message object as its first argument, and any arguments or keywords that were passed to it as additional arguments. By default `content_manager` is set to `raw_data_manager`.

Nouveau dans la version 3.4.

The class provides the following concrete implementations of the abstract methods of `Policy` :

header_max_count (*name*)

Returns the value of the `max_count` attribute of the specialized class used to represent the header with the given name.

header_source_parse (*sourcelines*)

The name is parsed as everything up to the “:” and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse (*name, value*)

The name is returned unchanged. If the input value has a `name` attribute and it matches `name` ignoring case, the value is returned unchanged. Otherwise the `name` and `value` are passed to `header_factory`, and the resulting header object is returned as the value. In this case a `ValueError` is raised if the input value contains CR or LF characters.

header_fetch_parse (*name, value*)

If the value has a `name` attribute, it is returned to unmodified. Otherwise the `name`, and the `value` with any CR or LF characters removed, are passed to the `header_factory`, and the resulting header object is returned. Any surrogateescaped bytes get turned into the unicode unknown-character glyph.

fold (*name, value*)

Header folding is controlled by the `refold_source` policy setting. A value is considered to be a “source value” if and only if it does not have a `name` attribute (having a `name` attribute means it is a header object of some sort). If a source value needs to be refolded according to the policy, it is converted into a header object by passing the `name` and the `value` with any CR and LF characters removed to the `header_factory`. Folding of a header object is done by calling its `fold` method with the current policy.

Source values are split into lines using `splitlines()`. If the value is not to be refolded, the lines are rejoined using the `linesep` from the policy and returned. The exception is lines containing non-ascii binary data. In that case the value is refolded regardless of the `refold_source` setting, which causes the binary data to be CTE encoded using the `unknown-8bit` charset.

fold_binary (*name, value*)

The same as `fold()` if `cte_type` is `7bit`, except that the returned value is bytes.

If `cte_type` is `8bit`, non-ASCII binary data is converted back into bytes. Headers with binary data are not refolded, regardless of the `refold_header` setting, since there is no way to know whether the binary

data consists of single byte characters or multibyte characters.

The following instances of *EmailPolicy* provide defaults suitable for specific application domains. Note that in the future the behavior of these instances (in particular the HTTP instance) may be adjusted to conform even more closely to the RFCs relevant to their domains.

`email.policy.default`

An instance of *EmailPolicy* with all defaults unchanged. This policy uses the standard Python `\n` line endings rather than the RFC-correct `\r\n`.

`email.policy.SMTP`

Suitable for serializing messages in conformance with the email RFCs. Like `default`, but with `linesep` set to `\r\n`, which is RFC compliant.

`email.policy.SMTPUTF8`

The same as `SMTP` except that `utf8` is `True`. Useful for serializing messages to a message store without using encoded words in the headers. Should only be used for SMTP transmission if the sender or recipient addresses have non-ASCII characters (the `smtplib.SMTP.send_message()` method handles this automatically).

`email.policy.HTTP`

Suitable for serializing headers with for use in HTTP traffic. Like `SMTP` except that `max_line_length` is set to `None` (unlimited).

`email.policy.strict`

Convenience instance. The same as `default` except that `raise_on_defect` is set to `True`. This allows any policy to be made strict by writing :

```
somepolicy + policy.strict
```

With all of these *EmailPolicies*, the effective API of the email package is changed from the Python 3.2 API in the following ways :

- Setting a header on a *Message* results in that header being parsed and a header object created.
- Fetching a header value from a *Message* results in that header being parsed and a header object created and returned.
- Any header object, or any header that is refolded due to the policy settings, is folded using an algorithm that fully implements the RFC folding algorithms, including knowing where encoded words are required and allowed.

From the application view, this means that any header obtained through the *EmailMessage* is a header object with extra attributes, whose string value is the fully decoded unicode value of the header. Likewise, a header may be assigned a new value, or a new header created, using a unicode string, and the policy will take care of converting the unicode string into the correct RFC encoded form.

The header objects and their attributes are described in *headerregistry*.

class `email.policy.Compat32` (**kw)

This concrete *Policy* is the backward compatibility policy. It replicates the behavior of the email package in Python 3.2. The *policy* module also defines an instance of this class, `compat32`, that is used as the default policy. Thus the default behavior of the email package is to maintain compatibility with Python 3.2.

The following attributes have values that are different from the *Policy* default :

mangle_from_

The default is `True`.

The class provides the following concrete implementations of the abstract methods of *Policy* :

header_source_parse (*sourcelines*)

The name is parsed as everything up to the “:” and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse (*name*, *value*)

The name and value are returned unmodified.

header_fetch_parse (*name*, *value*)

If the value contains binary data, it is converted into a *Header* object using the `unknown-8bit` charset. Otherwise it is returned unmodified.

fold (*name*, *value*)

Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. Non-ASCII binary data are CTE encoded using the `unknown-8bit` charset.

fold_binary (*name*, *value*)

Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. If `cte_type` is `7bit`, non-ascii binary data is CTE encoded using the `unknown-8bit` charset. Otherwise the original source header is used, with its existing line breaks and any (RFC invalid) binary data it may contain.

`email.policy.compat32`

An instance of *Compat32*, providing backward compatibility with the behavior of the email package in Python 3.2.

Notes

19.1.5 email.errors : Exception and Defect classes

Source code : <Lib/email/errors.py>

The following exception classes are defined in the *email.errors* module :

exception `email.errors.MessageError`

This is the base class for all exceptions that the *email* package can raise. It is derived from the standard *Exception* class and defines no additional methods.

exception `email.errors.MessageParseError`

This is the base class for exceptions raised by the *Parser* class. It is derived from *MessageError*. This class is also used internally by the parser used by *headerregistry*.

exception `email.errors.HeaderParseError`

Raised under some error conditions when parsing the **RFC 5322** headers of a message, this class is derived from *MessageParseError*. The *set_boundary()* method will raise this error if the content type is unknown when the method is called. *Header* may raise this error for certain base64 decoding errors, and when an attempt is made to create a header that appears to contain an embedded header (that is, there is what is supposed to be a continuation line that has no leading whitespace and looks like a header).

exception `email.errors.BoundaryError`

Deprecated and no longer used.

exception `email.errors.MultipartConversionError`

Raised when a payload is added to a *Message* object using *add_payload()*, but the payload is already a scalar and the message's *Content-Type* main type is not either *multipart* or missing. *MultipartConversionError* multiply inherits from *MessageError* and the built-in *TypeError*.

Since *Message.add_payload()* is deprecated, this exception is rarely raised in practice. However the exception may also be raised if the *attach()* method is called on an instance of a class derived from *MIMENonMultipart* (e.g. *MIMEImage*).

Here is the list of the defects that the *FeedParser* can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a *multipart/alternative* had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from `email.errors.MessageDefect`.

- `NoBoundaryInMultipartDefect` – A message claimed to be a multipart, but had no *boundary* parameter.
- `StartBoundaryNotFoundDefect` – The start boundary claimed in the *Content-Type* header was never found.
- `CloseBoundaryNotFoundDefect` – A start boundary was found, but no corresponding close boundary was ever found.
Nouveau dans la version 3.3.
- `FirstHeaderLineIsContinuationDefect` – The message had a continuation line as its first header line.
- `MisplacedEnvelopeHeaderDefect` – A « Unix From » header was found in the middle of a header block.
- `MissingHeaderBodySeparatorDefect` – A line was found while parsing headers that had no leading white space but contained no “:”. Parsing continues assuming that the line represents the first line of the body.
Nouveau dans la version 3.3.
- `MalformedHeaderDefect` – A header was found that was missing a colon, or was otherwise malformed.
Obsolète depuis la version 3.3 : This defect has not been used for several Python versions.
- `MultipartInvariantViolationDefect` – A message claimed to be a *multipart*, but no subparts were found. Note that when a message has this defect, its `is_multipart()` method may return false even though its content type claims to be *multipart*.
- `InvalidBase64PaddingDefect` – When decoding a block of base64 encoded bytes, the padding was not correct. Enough padding is added to perform the decode, but the resulting decoded bytes may be invalid.
- `InvalidBase64CharactersDefect` – When decoding a block of base64 encoded bytes, characters outside the base64 alphabet were encountered. The characters are ignored, but the resulting decoded bytes may be invalid.
- `InvalidBase64LengthDefect` – When decoding a block of base64 encoded bytes, the number of non-padding base64 characters was invalid (1 more than a multiple of 4). The encoded block was kept as-is.

19.1.6 `email.headerregistry` : Custom Header Objects

Source code : [Lib/email/headerregistry.py](#)

Nouveau dans la version 3.6 :¹

Headers are represented by customized subclasses of `str`. The particular class used to represent a given header is determined by the `header_factory` of the `policy` in effect when the headers are created. This section documents the particular `header_factory` implemented by the email package for handling **RFC 5322** compliant email messages, which not only provides customized header objects for various header types, but also provides an extension mechanism for applications to add their own custom header types.

When using any of the policy objects derived from `EmailPolicy`, all headers are produced by `HeaderRegistry` and have `BaseHeader` as their last base class. Each header class has an additional base class that is determined by the type of the header. For example, many headers have the class `UnstructuredHeader` as their other base class. The specialized second class for a header is determined by the name of the header, using a lookup table stored in the `HeaderRegistry`. All of this is managed transparently for the typical application program, but interfaces are provided for modifying the default behavior for use by more complex applications.

The sections below first document the header base classes and their attributes, followed by the API for modifying the behavior of `HeaderRegistry`, and finally the support classes used to represent the data parsed from structured headers.

class `email.headerregistry.BaseHeader` (*name*, *value*)

name and *value* are passed to `BaseHeader` from the `header_factory` call. The string value of any header object is the *value* fully decoded to unicode.

This base class defines the following read-only properties :

1. Originally added in 3.3 as a *provisional module*

name

The name of the header (the portion of the field before the “:”). This is exactly the value passed in the `header_factory` call for *name*; that is, case is preserved.

defects

A tuple of `HeaderDefect` instances reporting any RFC compliance problems found during parsing. The email package tries to be complete about detecting compliance issues. See the `errors` module for a discussion of the types of defects that may be reported.

max_count

The maximum number of headers of this type that can have the same *name*. A value of `None` means unlimited. The `BaseHeader` value for this attribute is `None`; it is expected that specialized header classes will override this value as needed.

`BaseHeader` also provides the following method, which is called by the email library code and should not in general be called by application programs :

fold (*, *policy*)

Return a string containing `linesep` characters as required to correctly fold the header according to *policy*. A `cte_type` of `8bit` will be treated as if it were `7bit`, since headers may not contain arbitrary binary data. If `utf8` is `False`, non-ASCII data will be **RFC 2047** encoded.

`BaseHeader` by itself cannot be used to create a header object. It defines a protocol that each specialized header cooperates with in order to produce the header object. Specifically, `BaseHeader` requires that the specialized class provide a `classmethod()` named `parse`. This method is called as follows :

```
parse(string, kwds)
```

`kwds` is a dictionary containing one pre-initialized key, `defects`. `defects` is an empty list. The `parse` method should append any detected defects to this list. On return, the `kwds` dictionary *must* contain values for at least the keys `decoded` and `defects`. `decoded` should be the string value for the header (that is, the header value fully decoded to unicode). The `parse` method should assume that *string* may contain content-transfer-encoded parts, but should correctly handle all valid unicode characters as well so that it can parse un-encoded header values.

`BaseHeader`’s `__new__` then creates the header instance, and calls its `init` method. The specialized class only needs to provide an `init` method if it wishes to set additional attributes beyond those provided by `BaseHeader` itself. Such an `init` method should look like this :

```
def init(self, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

That is, anything extra that the specialized class puts in to the `kwds` dictionary should be removed and handled, and the remaining contents of `kw` (and `args`) passed to the `BaseHeader` `init` method.

class email.headerregistry.**UnstructuredHeader**

An « unstructured » header is the default type of header in **RFC 5322**. Any header that does not have a specified syntax is treated as unstructured. The classic example of an unstructured header is the *Subject* header.

In **RFC 5322**, an unstructured header is a run of arbitrary text in the ASCII character set. **RFC 2047**, however, has an **RFC 5322** compatible mechanism for encoding non-ASCII text as ASCII characters within a header value. When a *value* containing encoded words is passed to the constructor, the `UnstructuredHeader` parser converts such encoded words into unicode, following the **RFC 2047** rules for unstructured text. The parser uses heuristics to attempt to decode certain non-compliant encoded words. Defects are registered in such cases, as well as defects for issues such as invalid characters within the encoded words or the non-encoded text.

This header type provides no additional attributes.

class email.headerregistry.**DateHeader**

RFC 5322 specifies a very specific format for dates within email headers. The `DateHeader` parser recognizes that date format, as well as recognizing a number of variant forms that are sometimes found « in the wild ».

This header type provides the following additional attributes :

datetime

If the header value can be recognized as a valid date of one form or another, this attribute will contain a

`datetime` instance representing that date. If the timezone of the input date is specified as `-0000` (indicating it is in UTC but contains no information about the source timezone), then `datetime` will be a naive `datetime`. If a specific timezone offset is found (including `+0000`), then `datetime` will contain an aware `datetime` that uses `datetime.timezone` to record the timezone offset.

The decoded value of the header is determined by formatting the `datetime` according to the **RFC 5322** rules; that is, it is set to :

```
email.utils.format_datetime(self.datetime)
```

When creating a `DateHeader`, `value` may be `datetime` instance. This means, for example, that the following code is valid and does what one would expect :

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

Because this is a naive `datetime` it will be interpreted as a UTC timestamp, and the resulting value will have a timezone of `-0000`. Much more useful is to use the `localtime()` function from the `utils` module :

```
msg['Date'] = utils.localtime()
```

This example sets the date header to the current time and date using the current timezone offset.

class email.headerregistry.AddressHeader

Address headers are one of the most complex structured header types. The `AddressHeader` class provides a generic interface to any address header.

This header type provides the following additional attributes :

groups

A tuple of `Group` objects encoding the addresses and groups found in the header value. Addresses that are not part of a group are represented in this list as single-address `Groups` whose `display_name` is `None`.

addresses

A tuple of `Address` objects encoding all of the individual addresses from the header value. If the header value contains any groups, the individual addresses from the group are included in the list at the point where the group occurs in the value (that is, the list of addresses is « flattened » into a one dimensional list).

The decoded value of the header will have all encoded words decoded to unicode. `idna` encoded domain names are also decoded to unicode. The decoded value is set by *joining* the `str` value of the elements of the `groups` attribute with `', '`.

A list of `Address` and `Group` objects in any combination may be used to set the value of an address header. `Group` objects whose `display_name` is `None` will be interpreted as single addresses, which allows an address list to be copied with groups intact by using the list obtained from the `groups` attribute of the source header.

class email.headerregistry.SingleAddressHeader

A subclass of `AddressHeader` that adds one additional attribute :

address

The single address encoded by the header value. If the header value actually contains more than one address (which would be a violation of the RFC under the default *policy*), accessing this attribute will result in a `ValueError`.

Many of the above classes also have a `Unique` variant (for example, `UniqueUnstructuredHeader`). The only difference is that in the `Unique` variant, `max_count` is set to 1.

class email.headerregistry.MIMEVersionHeader

There is really only one valid value for the `MIME-Version` header, and that is `1.0`. For future proofing, this header class supports other valid version numbers. If a version number has a valid value per **RFC 2045**, then the header object will have non-`None` values for the following attributes :

version

The version number as a string, with any whitespace and/or comments removed.

major

The major version number as an integer

minor

The minor version number as an integer

class email.headerregistry.**ParameterizedMIMEHeader**

MIME headers all start with the prefix “Content-“. Each specific header has a certain value, described under the class for that header. Some can also take a list of supplemental parameters, which have a common format. This class serves as a base for all the MIME headers that take parameters.

params

A dictionary mapping parameter names to parameter values.

class email.headerregistry.**ContentTypeHeader**

A *ParameterizedMIMEHeader* class that handles the *Content-Type* header.

content_type

The content type string, in the form maintype/subtype.

maintype

subtype

class email.headerregistry.**ContentDispositionHeader**

A *ParameterizedMIMEHeader* class that handles the *Content-Disposition* header.

content-disposition

inline and attachment are the only valid values in common use.

class email.headerregistry.**ContentTransferEncoding**

Handles the *Content-Transfer-Encoding* header.

cte

Valid values are 7bit, 8bit, base64, and quoted-printable. See [RFC 2045](#) for more information.

class email.headerregistry.**HeaderRegistry** (*base_class=BaseHeader*, *de-*
fault_class=UnstructuredHeader,
use_default_map=True)

This is the factory used by *EmailPolicy* by default. *HeaderRegistry* builds the class used to create a header instance dynamically, using *base_class* and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by *default_class* is used as the specialized class. When *use_default_map* is *True* (the default), the standard mapping of header names to classes is copied in to the registry during initialization. *base_class* is always the last class in the generated class’s `__bases__` list.

The default mappings are :

subject UniqueUnstructuredHeader

date UniqueDateHeader

resent-date DateHeader

orig-date UniqueDateHeader

sender UniqueSingleAddressHeader

resent-sender SingleAddressHeader

to UniqueAddressHeader

resent-to AddressHeader

cc UniqueAddressHeader

resent-cc AddressHeader

from UniqueAddressHeader

resent-from AddressHeader

reply-to UniqueAddressHeader

HeaderRegistry has the following methods :

map_to_type (*self*, *name*, *cls*)

name is the name of the header to be mapped. It will be converted to lower case in the registry. *cls* is the specialized class to be used, along with *base_class*, to create the class used to instantiate headers that match *name*.

__getitem__ (*name*)

Construct and return a class to handle creating a *name* header.

__call__ (*name*, *value*)

Retrieves the specialized header associated with *name* from the registry (using *default_class* if *name* does not appear in the registry) and composes it with *base_class* to produce a class, calls the constructed class's constructor, passing it the same argument list, and finally returns the class instance created thereby.

The following classes are the classes used to represent data parsed from structured headers and can, in general, be used by an application program to construct structured values to assign to specific headers.

class email.headerregistry.**Address** (*display_name*=", *username*", *domain*",
addr_spec=None)

The class used to represent an email address. The general form of an address is :

```
[display_name] <username@domain>
```

ou :

```
username@domain
```

where each part must conform to specific syntax rules spelled out in [RFC 5322](#).

As a convenience *addr_spec* can be specified instead of *username* and *domain*, in which case *username* and *domain* will be parsed from the *addr_spec*. An *addr_spec* must be a properly RFC quoted string; if it is not *Address* will raise an error. Unicode characters are allowed and will be property encoded when serialized. However, per the RFCs, unicode is *not* allowed in the username portion of the address.

display_name

The display name portion of the address, if any, with all quoting removed. If the address does not have a display name, this attribute will be an empty string.

username

The username portion of the address, with all quoting removed.

domain

The domain portion of the address.

addr_spec

The username@domain portion of the address, correctly quoted for use as a bare address (the second form shown above). This attribute is not mutable.

__str__ ()

The *str* value of the object is the address quoted according to [RFC 5322](#) rules, but with no Content Transfer Encoding of any non-ASCII characters.

To support SMTP ([RFC 5321](#)), *Address* handles one special case : if *username* and *domain* are both the empty string (or *None*), then the string value of the *Address* is <>.

class email.headerregistry.**Group** (*display_name*=None, *addresses*=None)

The class used to represent an address group. The general form of an address group is :

```
display_name: [address-list];
```

As a convenience for processing lists of addresses that consist of a mixture of groups and single addresses, a *Group* may also be used to represent single addresses that are not part of a group by setting *display_name* to *None* and providing a list of the single address as *addresses*.

display_name

The *display_name* of the group. If it is *None* and there is exactly one *Address* in *addresses*, then the *Group* represents a single address that is not in a group.

addresses

A possibly empty tuple of [Address](#) objects representing the addresses in the group.

__str__()

The `str` value of a `Group` is formatted according to [RFC 5322](#), but with no Content Transfer Encoding of any non-ASCII characters. If `display_name` is `None` and there is a single `Address` in the `addresses` list, the `str` value will be the same as the `str` of that single `Address`.

Notes**19.1.7 email.contentmanager : Managing MIME Content**

Source code : [Lib/email/contentmanager.py](#)

Nouveau dans la version 3.6 : ¹

class email.contentmanager.ContentManager

Base class for content managers. Provides the standard registry mechanisms to register converters between MIME content and other representations, as well as the `get_content` and `set_content` dispatch methods.

get_content (*msg*, **args*, ***kw*)

Look up a handler function based on the `mimetype` of *msg* (see next paragraph), call it, passing through all arguments, and return the result of the call. The expectation is that the handler will extract the payload from *msg* and return an object that encodes information about the extracted data.

To find the handler, look for the following keys in the registry, stopping with the first one found :

- the string representing the full MIME type (`maintype/subtype`)
- the string representing the `maintype`
- the empty string

If none of these keys produce a handler, raise a [KeyError](#) for the full MIME type.

set_content (*msg*, *obj*, **args*, ***kw*)

If the `maintype` is `multipart`, raise a [TypeError](#); otherwise look up a handler function based on the type of *obj* (see next paragraph), call `clear_content()` on the *msg*, and call the handler function, passing through all arguments. The expectation is that the handler will transform and store *obj* into *msg*, possibly making other changes to *msg* as well, such as adding various MIME headers to encode information needed to interpret the stored data.

To find the handler, obtain the type of *obj* (`typ = type(obj)`), and look for the following keys in the registry, stopping with the first one found :

- the type itself (`typ`)
- the type's fully qualified name (`typ.__module__ + '.' + typ.__qualname__`).
- the type's `qualname` (`typ.__qualname__`)
- the type's name (`typ.__name__`).

If none of the above match, repeat all of the checks above for each of the types in the *MRO* (`typ.__mro__`). Finally, if no other key yields a handler, check for a handler for the key `None`. If there is no handler for `None`, raise a [KeyError](#) for the fully qualified name of the type.

Also add a *MIME-Version* header if one is not present (see also [MIMEPart](#)).

add_get_handler (*key*, *handler*)

Record the function *handler* as the handler for *key*. For the possible values of *key*, see `get_content()`.

add_set_handler (*typekey*, *handler*)

Record *handler* as the function to call when an object of a type matching *typekey* is passed to `set_content()`. For the possible values of *typekey*, see `set_content()`.

1. Originally added in 3.4 as a *provisional module*

Content Manager Instances

Currently the email package provides only one concrete content manager, `raw_data_manager`, although more may be added in the future. `raw_data_manager` is the `content_manager` provided by `EmailPolicy` and its derivatives.

`email.contentmanager.raw_data_manager`

This content manager provides only a minimum interface beyond that provided by `Message` itself : it deals only with text, raw byte strings, and `Message` objects. Nevertheless, it provides significant advantages compared to the base API : `get_content` on a text part will return a unicode string without the application needing to manually decode it, `set_content` provides a rich set of options for controlling the headers added to a part and controlling the content transfer encoding, and it enables the use of the various `add_` methods, thereby simplifying the creation of multipart messages.

`email.contentmanager.get_content(msg, errors='replace')`

Return the payload of the part as either a string (for text parts), an `EmailMessage` object (for message/rfc822 parts), or a bytes object (for all other non-multipart types). Raise a `KeyError` if called on a multipart. If the part is a text part and `errors` is specified, use it as the error handler when decoding the payload to unicode. The default error handler is `replace`.

`email.contentmanager.set_content(msg, <str>, subtype="plain", charset='utf-8' cte=None, disposition=None, filename=None, cid=None, params=None, headers=None)`

`email.contentmanager.set_content(msg, <bytes>, maintype, subtype, cte="base64", disposition=None, filename=None, cid=None, params=None, headers=None)`

`email.contentmanager.set_content(msg, <EmailMessage>, cte=None, disposition=None, filename=None, cid=None, params=None, headers=None)`

Add headers and payload to `msg` :

Add a *Content-Type* header with a maintype/subtype value.

- For `str`, set the MIME maintype to `text`, and set the subtype to `subtype` if it is specified, or `plain` if it is not.
- For `bytes`, use the specified `maintype` and `subtype`, or raise a `TypeError` if they are not specified.
- For `EmailMessage` objects, set the maintype to `message`, and set the subtype to `subtype` if it is specified or `rfc822` if it is not. If `subtype` is `partial`, raise an error (bytes objects must be used to construct message/partial parts).

If `charset` is provided (which is valid only for `str`), encode the string to bytes using the specified character set. The default is `utf-8`. If the specified `charset` is a known alias for a standard MIME charset name, use the standard charset instead.

If `cte` is set, encode the payload using the specified content transfer encoding, and set the *Content-Transfer-Encoding* header to that value. Possible values for `cte` are `quoted-printable`, `base64`, `7bit`, `8bit`, and `binary`. If the input cannot be encoded in the specified encoding (for example, specifying a `cte` of `7bit` for an input that contains non-ASCII values), raise a `ValueError`.

- For `str` objects, if `cte` is not set use heuristics to determine the most compact encoding.
- For `EmailMessage`, per **RFC 2046**, raise an error if a `cte` of `quoted-printable` or `base64` is requested for `subtype rfc822`, and for any `cte` other than `7bit` for `subtype external-body`. For `message/rfc822`, use `8bit` if `cte` is not specified. For all other values of `subtype`, use `7bit`.

Note : A `cte` of `binary` does not actually work correctly yet. The `EmailMessage` object as modified by `set_content` is correct, but `BytesGenerator` does not serialize it correctly.

If `disposition` is set, use it as the value of the *Content-Disposition* header. If not specified, and `filename` is specified, add the header with the value `attachment`. If `disposition` is not specified and `filename` is also not specified, do not add the header. The only valid values for `disposition` are `attachment` and `inline`.

If `filename` is specified, use it as the value of the `filename` parameter of the *Content-Disposition* header.

If *cid* is specified, add a *Content-ID* header with *cid* as its value.

If *params* is specified, iterate its *items* method and use the resulting (key, value) pairs to set additional parameters on the *Content-Type* header.

If *headers* is specified and is a list of strings of the form *headervalue* or a list of header objects (distinguished from strings by having a *name* attribute), add the headers to *msg*.

Notes

19.1.8 email: Exemples

Cette page contient quelques exemples de l'utilisation du package *email* pour lire, écrire, et envoyer de simples messages mail, ainsi que des messages MIME plus complexes.

Premièrement, regardons comment créer et envoyer un message avec simplement du texte (le contenu textuel et les adresses peuvent tous deux contenir des caractères Unicodes) :

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

Analyser des entêtes **RFC 822** peut être aisément réalisé en utilisant les classes du module *parser* :

```
# Import the email modules we'll need
from email.parser import BytesParser, Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

# Or for parsing headers in a string (this is an uncommon operation), use:
headers = Parser(policy=default).parsestr(
    'From: Foo Bar <user@example.com>\n'
    'To: <someone_else@example.com>\n'
    'Subject: Test message\n'
    '\n'
    'Body would go here\n')
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

Voici un exemple de l'envoi d'un message MIME contenant une série de photos de famille qui sont stockés ensemble dans un dossier :

```
# Import smtplib for the actual sending function
import smtplib

# And imghdr to find the types of our images
import imghdr

# Here are the email package modules we'll need
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'Our family reunion'

# Open the files in binary mode. Use imghdr to figure out the
# MIME subtype for each specific image.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
        msg.add_attachment(img_data, maintype='image',
                           subtype=imghdr.what(None, img_data))

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

Voici un exemple d'envoi du contenu d'un dossier entier en tant que message mail :¹

```
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from argparse import ArgumentParser
```

(suite sur la page suivante)

1. Merci à Matthew Dixon Cowles pour l'inspiration originale et les exemples.


```

from email.message import EmailMessage
from email.policy import SMTP

def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process. Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
otherwise use the current directory. Only the regular
files in the directory are sent, and we don't recurse to
subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')

    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = 'Contents of directory %s' % os.path.abspath(directory)
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension. Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                              maintype=maintype,
                              subtype=subtype,
                              filename=filename)

    # Now send or store the message

```


(suite de la page précédente)

```

if args.output:
    with open(args.output, 'wb') as fp:
        fp.write(msg.as_bytes(policy=SMTP))
else:
    with smtplib.SMTP('localhost') as s:
        s.send_message(msg)

if __name__ == '__main__':
    main()

```

Voici un message de comment décomposer un message MIME comme celui ci dessus en tant que fichiers dans un dossier :

```

#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser

def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension

```

(suite sur la page suivante)

(suite de la page précédente)

```

        ext = '.bin'
        filename = 'part-%03d%s' % (counter, ext)
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))

if __name__ == '__main__':
    main()

```

Voici un exemple de création d'un message HTML avec une version en texte comme alternative. Pour rendre les choses un peu plus intéressantes, nous incluons aussi une image dans la partie HTML, nous sauvons une copie du message sur le disque, et nous l'envoyons.

```

#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Ayons asperges pour le déjeuner"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
            Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cela ressemble à un excellent recipie[1] déjeuner.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# Add the html version. This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cela ressemble à un excellent
      <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
        recipie
      </a> déjeuner.
    </p>
    
  </body>
</html>
""").format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')

```

(suite sur la page suivante)

(suite de la page précédente)

```

# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)

```

Si on nous avait envoyé le message de l'exemple précédent, voici la manière avec laquelle nous pourrions le traiter :

```

import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser

# An imaginary module that would make this work and be safe.
from imaginary import magic_html_parser

# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
# the least formatted payload is and print the first three lines. Of course,
# if the message has no plain text part printing the first three lines of html
# is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':

```

(suite sur la page suivante)

(suite de la page précédente)

```

    for line in richest.get_content().splitlines():
        print(line)
    sys.exit()
elif richest['content-type'].subtype == 'html':
    body = richest
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form of cid to html form.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    # The magic_html_parser has to rewrite the href="cid:..." attributes to
    # point to the filenames in partfiles. It also has to do a safety-sanitize
    # of the html. It could be written using html.parser.
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.

```

La sortie textuelle du code ci dessus est :

```

To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Ayons asperges pour le déjeuner

Salut!

Cela ressemble à un excellent recipie[1] déjeuner.

```

Notes

API héritée :

19.1.9 `email.message.Message` : Representing an email message using the `compat32` API

The `Message` class is very similar to the `EmailMessage` class, without the methods added by that class, and with the default behavior of certain other methods being slightly different. We also document here some methods that, while supported by the `EmailMessage` class, are not recommended unless you are dealing with legacy code.

The philosophy and structure of the two classes is otherwise the same.

This document describes the behavior under the default (for `Message`) policy `Compat32`. If you are going to use another policy, you should be using the `EmailMessage` class instead.

An email message consists of *headers* and a *payload*. Headers must be [RFC 5233](#) style names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as `multipart/*` or `message/rfc822`.

The conceptual model provided by a `Message` object is that of an ordered dictionary of headers with additional methods for accessing both specialized information from the headers, for accessing the payload, for generating a serialized version of the message, and for recursively walking over the object tree. Note that duplicate headers are supported but special methods must be used to access them.

The `Message` pseudo-dictionary is indexed by the header names, which must be ASCII values. The values of the dictionary are strings that are supposed to contain only ASCII characters; there is some special handling for non-ASCII input, but it doesn't always produce the correct results. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. There may also be a single envelope header, also known as the *Unix-From* header or the `From_` header. The *payload* is either a string or bytes, in the case of simple message objects, or a list of `Message` objects, for MIME container documents (e.g. `multipart/*` and `message/rfc822`).

Here are the methods of the `Message` class :

class `email.message.Message` (*policy=compat32*)

If *policy* is specified (it must be an instance of a *policy* class) use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the `compat32` policy, which maintains backward compatibility with the Python 3.2 version of the email package. For more information see the *policy* documentation.

Modifié dans la version 3.3 : The *policy* keyword argument was added.

as_string (*unixfrom=False, maxheaderlen=0, policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility reasons, *maxheaderlen* defaults to 0, so if you want a different value you must override it explicitly (the value specified for *max_line_length* in the policy will be ignored by this method). The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `Generator`.

Flattening the message may trigger changes to the `Message` if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the unix mbox format. For more flexibility, instantiate a `Generator` instance and use its `flatten()` method directly. For example :

```

from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()

```

If the message object contains binary data that is not encoded according to RFC standards, the non-compliant data will be replaced by unicode « unknown character » code points. (See also `as_bytes()` and `BytesGenerator`.)

Modifié dans la version 3.4 : the *policy* keyword argument was added.

`__str__()`

Equivalent to `as_string()`. Allows `str(msg)` to produce a string containing the formatted message.

`as_bytes(unixfrom=False, policy=None)`

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `BytesGenerator`.

Flattening the message may trigger changes to the *Message* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the unix mbox format. For more flexibility, instantiate a `BytesGenerator` instance and use its `flatten()` method directly. For example :

```

from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()

```

Nouveau dans la version 3.4.

`__bytes__()`

Equivalent to `as_bytes()`. Allows `bytes(msg)` to produce a bytes object containing the formatted message.

Nouveau dans la version 3.4.

`is_multipart()`

Return `True` if the message's payload is a list of sub-*Message* objects, otherwise return `False`. When `is_multipart()` returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). (Note that `is_multipart()` returning `True` does not necessarily mean that « `msg.get_content_maintype() == "multipart"` » will return the `True`. For example, `is_multipart` will return `True` when the *Message* is of type `message/rfc822`.)

`set_unixfrom(unixfrom)`

Set the message's envelope header to *unixfrom*, which should be a string.

`get_unixfrom()`

Return the message's envelope header. Defaults to `None` if the envelope header was never set.

`attach(payload)`

Add the given *payload* to the current payload, which must be `None` or a list of *Message* objects before the call. After the call, the payload will always be a list of *Message* objects. If you want to set the payload to a scalar object (e.g. a string), use `set_payload()` instead.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by `set_content()` and the related `make` and `add` methods.

get_payload (*i=None, decode=False*)

Return the current payload, which will be a list of *Message* objects when *is_multipart()* is True, or a string when *is_multipart()* is False. If the payload is a list and you mutate the list object, you modify the message's payload in place.

With optional argument *i*, *get_payload()* will return the *i*-th element of the payload, counting from zero, if *is_multipart()* is True. An *IndexError* will be raised if *i* is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. *is_multipart()* is False) and *i* is given, a *TypeError* is raised.

Optional *decode* is a flag indicating whether the payload should be decoded or not, according to the *Content-Transfer-Encoding* header. When True and the message is not a multipart, the payload will be decoded if this header's value is quoted-printable or base64. If some other encoding is used, or *Content-Transfer-Encoding* header is missing, the payload is returned as-is (undecoded). In all cases the returned value is binary data. If the message is a multipart and the *decode* flag is True, then None is returned. If the payload is base64 and it was not perfectly formed (missing padding, characters outside the base64 alphabet), then an appropriate defect will be added to the message's defect property (*InvalidBase64PaddingDefect* or *InvalidBase64CharactersDefect*, respectively).

When *decode* is False (the default) the body is returned as a string without decoding the *Content-Transfer-Encoding*. However, for a *Content-Transfer-Encoding* of 8bit, an attempt is made to decode the original bytes using the charset specified by the *Content-Type* header, using the replace error handler. If no charset is specified, or if the charset given is not recognized by the email package, the body is decoded using the default ASCII charset.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by *get_content()* and *iter_parts()*.

set_payload (*payload, charset=None*)

Set the entire message object's payload to *payload*. It is the client's responsibility to ensure the payload invariants. Optional *charset* sets the message's default character set; see *set_charset()* for details.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by *set_content()*.

set_charset (*charset*)

Set the character set of the payload to *charset*, which can either be a *Charset* instance (see *email.charset*), a string naming a character set, or None. If it is a string, it will be converted to a *Charset* instance. If *charset* is None, the *charset* parameter will be removed from the *Content-Type* header (the message will not be otherwise modified). Anything else will generate a *TypeError*.

If there is no existing *MIME-Version* header one will be added. If there is no existing *Content-Type* header, one will be added with a value of *text/plain*. Whether the *Content-Type* header already exists or not, its *charset* parameter will be set to *charset.output_charset*. If *charset.input_charset* and *charset.output_charset* differ, the payload will be re-encoded to the *output_charset*. If there is no existing *Content-Transfer-Encoding* header, then the payload will be transfer-encoded, if needed, using the specified *Charset*, and a header with the appropriate value will be added. If a *Content-Transfer-Encoding* header already exists, the payload is assumed to already be correctly encoded using that *Content-Transfer-Encoding* and is not modified.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by the *charset* parameter of the *email.message.EmailMessage.set_content()* method.

get_charset ()

Return the *Charset* instance associated with the message's payload.

This is a legacy method. On the *EmailMessage* class it always returns None.

The following methods implement a mapping-like interface for accessing the message's **RFC 2822** headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in a *Message* object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as *Header* objects with a charset of *unknown-8bit*.

__len__ ()

Return the total number of headers, including duplicates.

__contains__ (name)

Return true if the message object has a field named *name*. Matching is done case-insensitively and *name* should not include the trailing colon. Used for the `in` operator, e.g. :

```
if 'message-id' in myMessage:
    print('Message-ID:', myMessage['message-id'])
```

__getitem__ (name)

Return the value of the named header field. *name* should not include the colon field separator. If the header is missing, `None` is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the `get_all()` method to get the values of all the extant named headers.

__setitem__ (name, val)

Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing fields.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g. :

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__ (name)

Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys ()

Return a list of all the message's header field names.

values ()

Return a list of all the message's field values.

items ()

Return a list of 2-tuples containing all the message's field headers and values.

get (name, failobj=None)

Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (defaults to `None`).

Here are some additional useful methods :

get_all (name, failobj=None)

Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header (_name, _value, **_params)

Extended header setting. This method is similar to `__setitem__()` except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added. If the value contains non-ASCII characters, it can be specified as a three tuple in the format (CHARSET, LANGUAGE, VALUE), where CHARSET is a string naming the charset to be used to encode the value, LANGUAGE can usually be set to `None` or the empty string (see [RFC 2231](#) for other possibilities), and VALUE is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII

characters, it is automatically encoded in **RFC 2231** format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

Here's an example :

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example with non-ASCII characters :

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename*="iso-8859-1'Fu%DFballer.ppt"
```

replace_header (*_name*, *_value*)

Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case. If no matching header was found, a `KeyError` is raised.

get_content_type ()

Return the message's content type. The returned string is coerced to lower case of the form *maintype/subtype*. If there was no *Content-Type* header in the message the default type as given by `get_default_type()` will be returned. Since according to **RFC 2045**, messages always have a default type, `get_content_type()` will always return a value.

RFC 2045 defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** mandates that the default type be *text/plain*.

get_content_maintype ()

Return the message's main content type. This is the *maintype* part of the string returned by `get_content_type()`.

get_content_subtype ()

Return the message's sub-content type. This is the *subtype* part of the string returned by `get_content_type()`.

get_default_type ()

Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type (*ctype*)

Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header.

get_params (*failobj=None*, *header='content-type'*, *unquote=True*)

Return the message's *Content-Type* parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the '=' sign. The left hand side of the '=' is the key, while the right hand side is the value. If there is no '=' sign in the parameter the value is the empty string, otherwise the value is as described in `get_param()` and is unquoted if optional *unquote* is `True` (the default).

Optional *failobj* is the object to return if there is no *Content-Type* header. Optional *header* is the header to search instead of *Content-Type*.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

get_param (*param*, *failobj=None*, *header='content-type'*, *unquote=True*)

Return the value of the *Content-Type* header's parameter *param* as a string. If the message has no *Content-Type* header or if there is no such parameter, then *failobj* is returned (defaults to `None`).

Optional *header* if given, specifies the message header to use instead of *Content-Type*.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was **RFC 2231** encoded. When it's a 3-tuple, the elements of the value are of the form (CHARSET, LANGUAGE, VALUE). Note that both CHARSET and LANGUAGE can be None, in which case you should consider VALUE to be encoded in the `us-ascii` charset. You can usually ignore LANGUAGE.

If your application doesn't care whether the parameter was encoded as in **RFC 2231**, you can collapse the parameter value by calling `email.utils.collapse_rfc2231_value()`, passing in the return value from `get_param()`. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn't. For example :

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the VALUE item in the 3-tuple) is always unquoted, unless *unquote* is set to `False`.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

set_param(*param*, *value*, *header*='Content-Type', *requote*=`True`, *charset*=`None`, *language*=`"`, *replace*=`False`)

Set a parameter in the *Content-Type* header. If the parameter already exists in the header, its value will be replaced with *value*. If the *Content-Type* header has not yet been defined for this message, it will be set to *text/plain* and the new parameter value will be appended as per **RFC 2045**.

Optional *header* specifies an alternative header to *Content-Type*, and all parameters will be quoted as necessary unless optional *requote* is `False` (the default is `True`).

If optional *charset* is specified, the parameter will be encoded according to **RFC 2231**. Optional *language* specifies the RFC 2231 language, defaulting to the empty string. Both *charset* and *language* should be strings.

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

Modifié dans la version 3.4 : *replace* keyword was added.

del_param(*param*, *header*='content-type', *requote*=`True`)

Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless *requote* is `False` (the default is `True`). Optional *header* specifies an alternative to *Content-Type*.

set_type(*type*, *header*='Content-Type', *requote*=`True`)

Set the main type and subtype for the *Content-Type* header. *type* must be a string in the form *maintype/subtype*, otherwise a `ValueError` is raised.

This method replaces the *Content-Type* header, keeping all the parameters in place. If *requote* is `False`, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the *header* argument. When the *Content-Type* header is set a *MIME-Version* header is also added.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `make_` and `add_` methods.

get_filename(*failobj*=`None`)

Return the value of the *filename* parameter of the *Content-Disposition* header of the message. If the header does not have a *filename* parameter, this method falls back to looking for the *name* parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per `email.utils.unquote()`.

get_boundary(*failobj*=`None`)

Return the value of the *boundary* parameter of the *Content-Type* header of the message, or *failobj* if either the header is missing, or has no *boundary* parameter. The returned string will always be unquoted as per `email.utils.unquote()`.

set_boundary (*boundary*)

Set the *boundary* parameter of the *Content-Type* header to *boundary*. *set_boundary()* will always quote *boundary* if necessary. A *HeaderParseError* is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different than deleting the old *Content-Type* header and adding a new one with the new boundary via *add_header()*, because *set_boundary()* preserves the order of the *Content-Type* header in the list of headers. However, it does *not* preserve any continuation lines which may have been present in the original *Content-Type* header.

get_content_charset (*failobj=None*)

Return the *charset* parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no *charset* parameter, *failobj* is returned.

Note that this method differs from *get_charset()* which returns the *Charset* instance for the default encoding of the message body.

get_charsets (*failobj=None*)

Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the *charset* parameter in the *Content-Type* header for the represented subpart. However, if the subpart has no *Content-Type* header, no *charset* parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

get_content_disposition ()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or *None*. The possible values for this method are *inline*, *attachment* or *None* if the message follows [RFC 2183](#).

Nouveau dans la version 3.5.

walk ()

The *walk()* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *walk()* as the iterator in a *for* loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure :

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

walk iterates over the subparts of any part where *is_multipart()* returns *True*, even though *msg.get_content_maintype() == 'multipart'* may return *False*. We can see this in our example by making use of the *_structure* debug helper function :

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart'),
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False True
False False
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> _structure(msg)
multipart/report
  text/plain
message/delivery-status
  text/plain
  text/plain
message/rfc822
  text/plain
```

Here the message parts are not multipart, but they do contain subparts. `is_multipart()` returns True and `walk` descends into the subparts.

Message objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

preamble

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the *Parser* discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the *Generator* is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See *email.parser* and *email.generator* for details.

Note that if the message object has no preamble, the *preamble* attribute will be None.

epilogue

The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message.

You do not need to set the epilogue to the empty string in order for the *Generator* to print a newline at the end of the file.

defects

The *defects* attribute contains a list of all the problems found when parsing this message. See *email.errors* for a detailed description of the possible parsing defects.

19.1.10 *email.mime* : Creating email and MIME objects from scratch

Source code : [Lib/email/mime/](#)

This module is part of the legacy (Compat32) email API. Its functionality is partially replaced by the *contentmanager* in the new API, but in certain applications these classes may still be useful, even in non-legacy code.

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual *Message* objects by hand. In fact, you can also take an existing structure and add new *Message* objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating *Message* instances, adding attachments and all the appropriate headers manually. For MIME messages though, the *email* package provides some convenient subclasses to make things easier.

Here are the classes :

```
class email.mime.base.MIMEBase(_maintype, _subtype, *, policy=compat32, **_params)
    Module: email.mime.base
```

This is the base class for all the MIME-specific subclasses of *Message*. Ordinarily you won't create instances specifically of *MIMEBase*, although you could. *MIMEBase* is provided primarily as a convenient base class for more specific MIME-aware subclasses.

_maintype is the *Content-Type* major type (e.g. *text* or *image*), and *_subtype* is the *Content-Type* minor type (e.g. *plain* or *gif*). *_params* is a parameter key/value dictionary and is passed directly to *Message.add_header*.

If *policy* is specified, (defaults to the *compat32* policy) it will be passed to *Message*.

The *MIMEBase* class always adds a *Content-Type* header (based on *_maintype*, *_subtype*, and *_params*), and a *MIME-Version* header (always set to 1.0).

Modifié dans la version 3.6 : Added *policy* keyword-only parameter.

```
class email.mime.nonmultipart.MIMENonMultipart
```

Module: email.mime.nonmultipart

A subclass of *MIMEBase*, this is an intermediate base class for MIME messages that are not *multipart*. The primary purpose of this class is to prevent the use of the *attach()* method, which only makes sense for *multipart* messages. If *attach()* is called, a *MultipartConversionError* exception is raised.

```
class email.mime.multipart.MIMEMultipart(_subtype='mixed', boundary=None, _sub-
                                         parts=None, *, policy=compat32, **_params)
```

Module: email.mime.multipart

A subclass of *MIMEBase*, this is an intermediate base class for MIME messages that are *multipart*. Optional *_subtype* defaults to *mixed*, but can be used to specify the subtype of the message. A *Content-Type* header of *multipart/_subtype* will be added to the message object. A *MIME-Version* header will also be added.

Optional *boundary* is the multipart boundary string. When *None* (the default), the boundary is calculated when needed (for example, when the message is serialized).

_subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the *Message.attach* method.

Optional *policy* argument defaults to *compat32*.

Additional parameters for the *Content-Type* header are taken from the keyword arguments, or passed into the *_params* argument, which is a keyword dictionary.

Modifié dans la version 3.6 : Added *policy* keyword-only parameter.

```
class email.mime.application.MIMEApplication(_data, _subtype='octet-stream', _enco-
                                             der=email.encoders.encode_base64, *,
                                             policy=compat32, **_params)
```

Module: email.mime.application

A subclass of *MIMENonMultipart*, the *MIMEApplication* class is used to represent MIME message objects of major type *application*. *_data* is a string containing the raw byte data. Optional *_subtype* specifies the MIME subtype and defaults to *octet-stream*.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the data for transport. This callable takes one argument, which is the *MIMEApplication* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

Optional *policy* argument defaults to *compat32*.

_params are passed straight through to the base class constructor.

Modifié dans la version 3.6 : Added *policy* keyword-only parameter.

```
class email.mime.audio.MIMEAudio(_audiodata, _subtype=None, _enco-
                                  der=email.encoders.encode_base64, *, policy=compat32,
                                  **_params)
```

Module: email.mime.audio

A subclass of *MIMENonMultipart*, the *MIMEAudio* class is used to create MIME message objects of major type *audio*. *_audiodata* is a string containing the raw audio data. If this data can be decoded by the standard Python

module `sndhdr`, then the subtype will be automatically included in the `Content-Type` header. Otherwise you can explicitly specify the audio subtype via the `_subtype` argument. If the minor type could not be guessed and `_subtype` was not given, then `TypeError` is raised.

Optional `_encoder` is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the `MIMEAudio` instance. It should use `get_payload()` and `set_payload()` to change the payload to encoded form. It should also add any `Content-Transfer-Encoding` or other headers to the message object as necessary. The default encoding is base64. See the `email.encoders` module for a list of the built-in encoders.

Optional `policy` argument defaults to `compat32`.

`_params` are passed straight through to the base class constructor.

Modifié dans la version 3.6 : Added `policy` keyword-only parameter.

```
class email.mime.image.MIMEImage(_imagedata, _subtype=None, _encoder=
email.encoders.encode_base64, *, policy=compat32,
**_params)
```

Module : `email.mime.image`

A subclass of `MIMENonMultipart`, the `MIMEImage` class is used to create MIME message objects of major type `image`. `_imagedata` is a string containing the raw image data. If this data can be decoded by the standard Python module `imghdr`, then the subtype will be automatically included in the `Content-Type` header. Otherwise you can explicitly specify the image subtype via the `_subtype` argument. If the minor type could not be guessed and `_subtype` was not given, then `TypeError` is raised.

Optional `_encoder` is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the `MIMEImage` instance. It should use `get_payload()` and `set_payload()` to change the payload to encoded form. It should also add any `Content-Transfer-Encoding` or other headers to the message object as necessary. The default encoding is base64. See the `email.encoders` module for a list of the built-in encoders.

Optional `policy` argument defaults to `compat32`.

`_params` are passed straight through to the `MIMEBase` constructor.

Modifié dans la version 3.6 : Added `policy` keyword-only parameter.

```
class email.mime.message.MIMEMessage(_msg, _subtype='rfc822', *, policy=compat32)
```

Module : `email.mime.message`

A subclass of `MIMENonMultipart`, the `MIMEMessage` class is used to create MIME objects of main type `message`. `_msg` is used as the payload, and must be an instance of class `Message` (or a subclass thereof), otherwise a `TypeError` is raised.

Optional `_subtype` sets the subtype of the message; it defaults to `rfc822`.

Optional `policy` argument defaults to `compat32`.

Modifié dans la version 3.6 : Added `policy` keyword-only parameter.

```
class email.mime.text.MIMEText(_text, _subtype='plain', _charset=None, *, policy=compat32)
```

Module : `email.mime.text`

A subclass of `MIMENonMultipart`, the `MIMEText` class is used to create MIME objects of major type `text`. `_text` is the string for the payload. `_subtype` is the minor type and defaults to `plain`. `_charset` is the character set of the text and is passed as an argument to the `MIMENonMultipart` constructor; it defaults to `us-ascii` if the string contains only `ascii` code points, and `utf-8` otherwise. The `_charset` parameter accepts either a string or a `Charset` instance.

Unless the `_charset` argument is explicitly set to `None`, the `MIMEText` object created will have both a `Content-Type` header with a `charset` parameter, and a `Content-Transfer-Encoding` header. This means that a subsequent `set_payload` call will not result in an encoded payload, even if a `charset` is passed in the `set_payload` command. You can « reset » this behavior by deleting the `Content-Transfer-Encoding` header, after which a `set_payload` call will automatically encode the new payload (and add a new `Content-Transfer-Encoding` header).

Optional `policy` argument defaults to `compat32`.

Modifié dans la version 3.5 : `_charset` also accepts `Charset` instances.

Modifié dans la version 3.6 : Added `policy` keyword-only parameter.

19.1.11 `email.header` : Internationalized headers

Source code : [Lib/email/header.py](#)

This module is part of the legacy (Compat32) email API. In the current API encoding and decoding of headers is handled transparently by the dictionary-like API of the `EmailMessage` class. In addition to uses in legacy code, this module can be useful in applications that need to completely control the character sets used when encoding headers.

Le texte restant de cette section est la documentation originale de ce module.

RFC 2822 is the base standard that describes the format of email messages. It derives from the older **RFC 822** standard which came into widespread use at a time when most email was composed of ASCII characters only. **RFC 2822** is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into **RFC 2822**-compliant format. These RFCs include **RFC 2045**, **RFC 2046**, **RFC 2047**, and **RFC 2231**. The `email` package supports these standards in its `email.header` and `email.charset` modules.

If you want to include non-ASCII characters in your email headers, say in the `Subject` or `To` fields, you should use the `Header` class and assign the field in the `Message` object to an instance of `Header` instead of using a string for the header value. Import the `Header` class from the `email.header` module. For example :

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xxf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

Notice here how we wanted the `Subject` field to contain a non-ASCII character? We did this by creating a `Header` instance and passing in the character set that the byte string was encoded in. When the subsequent `Message` instance was flattened, the `Subject` field was properly **RFC 2047** encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

Here is the `Header` class description :

```
class email.header.Header (s=None, charset=None, maxlinelen=None, header_name=None, continuation_ws=' ', errors='strict')
```

Create a MIME-compliant header that can contain strings in different character sets.

Optional `s` is the initial header value. If `None` (the default), the initial header value is not set. You can later append to the header with `append()` method calls. `s` may be an instance of `bytes` or `str`, but see the `append()` documentation for semantics.

Optional `charset` serves two purposes : it has the same meaning as the `charset` argument to the `append()` method. It also sets the default character set for all subsequent `append()` calls that omit the `charset` argument. If `charset` is not provided in the constructor (the default), the `us-ascii` character set is used both as `s`'s initial charset and as the default for subsequent `append()` calls.

The maximum line length can be specified explicitly via `maxlinelen`. For splitting the first line to a shorter value (to account for the field header which isn't included in `s`, e.g. `Subject`) pass in the name of the field in `header_name`. The default `maxlinelen` is 76, and the default value for `header_name` is `None`, meaning it is not taken into account for the first line of a long, split header.

Optional *continuation_ws* must be [RFC 2822](#)-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. *continuation_ws* defaults to a single space character.

Optional *errors* is passed straight through to the *append()* method.

append (*s*, *charset=None*, *errors='strict'*)

Append the string *s* to the MIME header.

Optional *charset*, if given, should be a *Charset* instance (see *email.charset*) or the name of a character set, which will be converted to a *Charset* instance. A value of *None* (the default) means that the *charset* given in the constructor is used.

s may be an instance of *bytes* or *str*. If it is an instance of *bytes*, then *charset* is the encoding of that byte string, and a *UnicodeError* will be raised if the string cannot be decoded with that character set.

If *s* is an instance of *str*, then *charset* is a hint specifying the character set of the characters in the string.

In either case, when producing an [RFC 2822](#)-compliant header using [RFC 2047](#) rules, the string will be encoded using the output codec of the charset. If the string cannot be encoded using the output codec, a *UnicodeError* will be raised.

Optional *errors* is passed as the *errors* argument to the decode call if *s* is a byte string.

encode (*splitchars='; \t'*, *maxlinelen=None*, *linesep='\n'*)

Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings.

Optional *splitchars* is a string containing characters which should be given extra weight by the splitting algorithm during normal header wrapping. This is in very rough support of [RFC 2822](#)'s "higher level syntactic breaks": split points preceded by a splitchar are preferred during line splitting, with the characters preferred in the order in which they appear in the string. Space and tab may be included in the string to indicate whether preference should be given to one over the other as a split point when other split chars do not appear in the line being split. Splitchars does not affect [RFC 2047](#) encoded lines.

maxlinelen, if given, overrides the instance's value for the maximum line length.

linesep specifies the characters used to separate the lines of the folded header. It defaults to the most useful value for Python application code (`\n`), but `\r\n` can be specified in order to produce headers with RFC-compliant line separators.

Modifié dans la version 3.2 : Added the *linesep* argument.

The *Header* class also provides a number of methods to support standard operators and built-in functions.

__str__ ()

Returns an approximation of the *Header* as a string, using an unlimited line length. All pieces are converted to unicode using the specified encoding and joined together appropriately. Any pieces with a charset of 'unknown-8bit' are decoded as ASCII using the 'replace' error handler.

Modifié dans la version 3.2 : Added handling for the 'unknown-8bit' charset.

__eq__ (*other*)

This method allows you to compare two *Header* instances for equality.

__ne__ (*other*)

This method allows you to compare two *Header* instances for inequality.

The *email.header* module also provides the following convenient functions.

email.header.decode_header (*header*)

Decode a message header value without converting the character set. The header value is in *header*.

This function returns a list of (*decoded_string*, *charset*) pairs containing each of the decoded parts of the header. *charset* is *None* for non-encoded parts of the header, otherwise a lower case string containing the name of the character set specified in the encoded string.

Here's an example :

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xxf6stal', 'iso-8859-1')]
```


`email.header.make_header` (*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=''*)

Create a *Header* instance from a sequence of pairs as returned by *decode_header()*.

decode_header() takes a header value string and returns a sequence of pairs of the format (*decoded_string*, *charset*) where *charset* is the name of the character set.

This function takes one of those sequence of pairs and returns a *Header* instance. Optional *maxlinelen*, *header_name*, and *continuation_ws* are as in the *Header* constructor.

19.1.12 email.charset : Representing character sets

Source code : <Lib/email/charset.py>

This module is part of the legacy (Compat32) email API. In the new API only the aliases table is used.

Le texte restant de cette section est la documentation originale de ce module.

This module provides a class *Charset* for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of *Charset* are used in several other modules within the *email* package.

Import this class from the *email.charset* module.

class `email.charset.Charset` (*input_charset=DEFAULT_CHARSET*)

Map character sets to their email properties.

This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.

Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional *input_charset* is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if *input_charset* is *iso-8859-1*, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If *input_charset* is *euc-jp*, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the *euc-jp* character set to the *iso-2022-jp* character set.

Charset instances have the following data attributes :

input_charset

The initial character set specified. Common aliases are converted to their *official* email names (e.g. *latin_1* is converted to *iso-8859-1*). Defaults to 7-bit *us-ascii*.

header_encoding

If the character set must be encoded before it can be used in an email header, this attribute will be set to *Charset.QP* (for quoted-printable), *Charset.BASE64* (for base64 encoding), or *Charset.SHORTEST* for the shortest of QP or BASE64 encoding. Otherwise, it will be *None*.

body_encoding

Same as *header_encoding*, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. *Charset.SHORTEST* is not allowed for *body_encoding*.

output_charset

Some character sets must be converted before they can be used in email headers or bodies. If the *input_charset* is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be *None*.

input_codec

The name of the Python codec used to convert the *input_charset* to Unicode. If no conversion codec is necessary, this attribute will be *None*.

output_codec

The name of the Python codec used to convert Unicode to the *output_charset*. If no conversion codec is necessary, this attribute will have the same value as the *input_codec*.

Charset instances also have the following methods :

get_body_encoding()

Return the content transfer encoding used for body encoding.

This is either the string *quoted-printable* or *base64* depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the *Content-Transfer-Encoding* header itself to whatever is appropriate.

Returns the string *quoted-printable* if *body_encoding* is *QP*, returns the string *base64* if *body_encoding* is *BASE64*, and returns the string *7bit* otherwise.

get_output_charset()

Return the output character set.

This is the *output_charset* attribute if that is not *None*, otherwise it is *input_charset*.

header_encode(string)

Header-encode the string *string*.

The type of encoding (*base64* or *quoted-printable*) will be based on the *header_encoding* attribute.

header_encode_lines(string, maxlengths)

Header-encode a *string* by converting it first to bytes.

This is similar to *header_encode()* except that the string is fit into maximum line lengths as given by the argument *maxlengths*, which must be an iterator : each element returned from this iterator will provide the next maximum line length.

body_encode(string)

Body-encode the string *string*.

The type of encoding (*base64* or *quoted-printable*) will be based on the *body_encoding* attribute.

The *Charset* class also provides a number of methods to support standard operations and built-in functions.

__str__()

Returns *input_charset* as a string coerced to lower case. *__repr__()* is an alias for *__str__()*.

__eq__(other)

This method allows you to compare two *Charset* instances for equality.

__ne__(other)

This method allows you to compare two *Charset* instances for inequality.

The *email.charset* module also provides the following functions for adding new entries to the global character set, alias, and codec registries :

email.charset.add_charset(charset, header_enc=None, body_enc=None, output_charset=None)

Add character properties to the global registry.

charset is the input character set, and must be the canonical name of a character set.

Optional *header_enc* and *body_enc* is either *Charset.QP* for quoted-printable, *Charset.BASE64* for base64 encoding, *Charset.SHORTEST* for the shortest of quoted-printable or base64 encoding, or *None* for no encoding. *SHORTEST* is only valid for *header_enc*. The default is *None* for no encoding.

Optional *output_charset* is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method *Charset.convert()* is called. The default is to output in the same character set as the input.

Both *input_charset* and *output_charset* must have Unicode codec entries in the module's character set-to-codec mapping; use *add_codec()* to add codecs the module does not know about. See the *codecs* module's documentation for more information.

The global character set registry is kept in the module global dictionary *CHARSETS*.

email.charset.add_alias(alias, canonical)

Add a character set alias. *alias* is the alias name, e.g. *latin-1*. *canonical* is the character set's canonical name, e.g. *iso-8859-1*.

The global charset alias registry is kept in the module global dictionary `ALIASES`.

`email.charset.add_codec(charset, codecname)`

Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. *codecname* is the name of a Python codec, as appropriate for the second argument to the *str*'s `encode()` method.

19.1.13 email.encoders : Encodeurs

Code source : [Lib/email/encoders.py](#)

Ce module fait partie du code patrimonial (Compat 32) de l'API mail. Dans la nouvelle API la fonctionnalité est fournie par le paramètre *cte* de la méthode `set_content()`.

Le texte restant de cette section est la documentation originale de ce module.

Au moment de la création d'objets *Message* à la main, il est souvent nécessaire d'encoder les charges utiles pour le transport à travers des serveurs mail conformes. C'est particulièrement vrai pour les messages de type *image/** et *text/** contenant des données binaires.

Le paquet *email* fournit quelques encodeurs pratiques dans son module `encoders`. Ces encodeurs sont d'ailleurs utilisés par les constructeurs des classes *MIMEAudio* et *MIMEImage* afin de fournir des encodages par défaut. Toutes les fonctions d'encodage prennent exactement un argument, l'objet message à encoder. Généralement, elles extraient la charge utile, l'encodent, puis change la charge utile pour la nouvelle valeur encodée. Elles devraient également assigner l'en-tête *Content-Transfer-Encoding* si besoin.

À noter que ces fonctions n'ont pas de sens dans le cadre d'un message en plusieurs parties. Elles doivent à la place être appliquées aux sous-parties individuelles, et lèvent *TypeError* si on leur passe un message en plusieurs parties.

Voici les fonctions d'encodages fournies :

`email.encoders.encode_quopri(msg)`

Encode la charge utile au format Quoted-Printable, et assigne `quoted-printable`¹ à l'en-tête *Content-Transfer-Encoding*. C'est un bon encodage à utiliser quand la majorité de la charge utile contient essentiellement des données imprimables, à l'exception de quelques caractères.

`email.encoders.encode_base64(msg)`

Encode la charge utile au format *base64*, et assigne *base64* à l'en-tête *Content-Transfer-Encoding*. C'est un bon encodage à utiliser quand la majorité de la charge utile est non imprimable puisque c'est une forme plus compacte que *quoted-printable*.

`email.encoders.encode_7or8bit(msg)`

Ceci ne modifie pas effectivement la charge utile du message, mais va bien en revanche assigner la valeur *7bit* ou *8bit* à l'en-tête *Content-Transfer-Encoding* selon la nature de la charge utile.

`email.encoders.encode_noop(msg)`

Ceci ne fait rien ; et ne va même pas changer la valeur de l'en-tête *Content-Transfer-Encoding*.

1. À noter que l'encodage avec `encode_quopri()` encode également tous les caractères tabulation et espace.

Notes

19.1.14 `email.utils` : Miscellaneous utilities

Source code : [Lib/email/utils.py](#)

There are a couple of useful utilities provided in the `email.utils` module :

`email.utils.localtime (dt=None)`

Return local time as an aware datetime object. If called without arguments, return current time. Otherwise *dt* argument should be a `datetime` instance, and it is converted to the local time zone according to the system time zone database. If *dt* is naive (that is, `dt.tzinfo` is `None`), it is assumed to be in local time. In this case, a positive or zero value for *isdst* causes `localtime` to presume initially that summer time (for example, Daylight Saving Time) is or is not (respectively) in effect for the specified time. A negative value for *isdst* causes the `localtime` to attempt to divine whether summer time is in effect for the specified time.

Nouveau dans la version 3.3.

`email.utils.make_msgid (idstring=None, domain=None)`

Returns a string suitable for an [RFC 2822](#)-compliant *Message-ID* header. Optional *idstring* if given, is a string used to strengthen the uniqueness of the message id. Optional *domain* if given provides the portion of the msgid after the “@”. The default is the local hostname. It is not normally necessary to override this default, but may be useful certain cases, such as a constructing distributed system that uses a consistent domain name across multiple hosts.

Modifié dans la version 3.2 : Added the *domain* keyword.

The remaining functions are part of the legacy (Compat32) email API. There is no need to directly use these with the new API, since the parsing and formatting they provide is done automatically by the header parsing machinery of the new API.

`email.utils.quote (str)`

Return a new string with backslashes in *str* replaced by two backslashes, and double quotes replaced by backslash-double quote.

`email.utils.unquote (str)`

Return a new string which is an *unquoted* version of *str*. If *str* ends and begins with double quotes, they are stripped off. Likewise if *str* ends and begins with angle brackets, they are stripped off.

`email.utils.parseaddr (address)`

Parse address – which should be the value of some address-containing field such as *To* or *Cc* – into its constituent *realname* and *email address* parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of (`''`, `''`) is returned.

`email.utils.formataddr (pair, charset='utf-8')`

The inverse of `parseaddr()`, this takes a 2-tuple of the form (*realname*, *email_address*) and returns the string value suitable for a *To* or *Cc* header. If the first element of *pair* is false, then the second element is returned unmodified.

Optional *charset* is the character set that will be used in the [RFC 2047](#) encoding of the *realname* if the *realname* contains non-ASCII characters. Can be an instance of *str* or a *Charset*. Defaults to `utf-8`.

Modifié dans la version 3.3 : Added the *charset* option.

`email.utils.getaddresses (fieldvalues)`

This method returns a list of 2-tuples of the form returned by `parseaddr()`. *fieldvalues* is a sequence of header field values as might be returned by `Message.get_all`. Here's a simple example that gets all the recipients of a message :

```

from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)

```

`email.utils.parsedate(date)`

Attempts to parse a date according to the rules in [RFC 2822](#), however, some mailers don't follow that format as specified, so `parsedate()` tries to guess correctly in such cases. `date` is a string containing an [RFC 2822](#) date, such as "Mon, 20 Nov 1995 19:12:08 -0500". If it succeeds in parsing the date, `parsedate()` returns a 9-tuple that can be passed directly to `time.mktime()`; otherwise `None` will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

`email.utils.parsedate_tz(date)`

Performs the same function as `parsedate()`, but returns either `None` or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to `time.mktime()`, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time)¹. If the input string has no timezone, the last element of the tuple returned is `None`. Note that indexes 6, 7, and 8 of the result tuple are not usable.

`email.utils.parsedate_to_datetime(date)`

The inverse of `format_datetime()`. Performs the same function as `parsedate()`, but on success returns a `datetime`. If the input date has a timezone of -0000, the `datetime` will be a naive `datetime`, and if the date is conforming to the RFCs it will represent a time in UTC but with no indication of the actual source timezone of the message the date comes from. If the input date has any other valid timezone offset, the `datetime` will be an aware `datetime` with the corresponding a `timezone.tzinfo`.

Nouveau dans la version 3.3.

`email.utils.mktime_tz(tuple)`

Turn a 10-tuple as returned by `parsedate_tz()` into a UTC timestamp (seconds since the Epoch). If the timezone item in the tuple is `None`, assume local time.

`email.utils.formatdate(timeval=None, localtime=False, usegmt=False)`

Returns a date string as per [RFC 2822](#), e.g. :

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional `timeval` if given is a floating point time value as accepted by `time.gmtime()` and `time.localtime()`, otherwise the current time is used.

Optional `localtime` is a flag that when `True`, interprets `timeval`, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is `False` meaning UTC is used.

Optional `usegmt` is a flag that when `True`, outputs a date string with the timezone as an ascii string GMT, rather than a numeric -0000. This is needed for some protocols (such as HTTP). This only applies when `localtime` is `False`. The default is `False`.

`email.utils.format_datetime(dt, usegmt=False)`

Like `formatdate`, but the input is a `datetime` instance. If it is a naive `datetime`, it is assumed to be « UTC with no information about the source timezone », and the conventional -0000 is used for the timezone. If it is an aware `datetime`, then the numeric timezone offset is used. If it is an aware timezone with offset zero, then `usegmt` may be set to `True`, in which case the string GMT is used instead of the numeric timezone offset. This provides a way to generate standards conformant HTTP date headers.

Nouveau dans la version 3.3.

1. Note that the sign of the timezone offset is the opposite of the sign of the `time.timezone` variable for the same timezone; the latter variable follows the POSIX standard while this module follows [RFC 2822](#).

`email.utils.decode_rfc2231(s)`

Decode the string *s* according to [RFC 2231](#).

`email.utils.encode_rfc2231(s, charset=None, language=None)`

Encode the string *s* according to [RFC 2231](#). Optional *charset* and *language*, if given is the character set name and language name to use. If neither is given, *s* is returned as-is. If *charset* is given but *language* is not, the string is encoded using the empty string for *language*.

`email.utils.collapse_rfc2231_value(value, errors='replace', fallback_charset='us-ascii')`

When a header parameter is encoded in [RFC 2231](#) format, `Message.get_param` may return a 3-tuple containing the character set, language, and value. `collapse_rfc2231_value()` turns this into a unicode string. Optional *errors* is passed to the *errors* argument of `str`'s `encode()` method; it defaults to 'replace'. Optional *fallback_charset* specifies the character set to use if the one in the [RFC 2231](#) header is not known by Python; it defaults to 'us-ascii'.

For convenience, if the *value* passed to `collapse_rfc2231_value()` is not a tuple, it should be a string and it is returned unquoted.

`email.utils.decode_params(params)`

Decode parameters list according to [RFC 2231](#). *params* is a sequence of 2-tuples containing elements of the form (content-type, string-value).

Notes

19.1.15 email.iterators : Itérateurs

Code source : [Lib/email/iterators.py](#)

Itérer sur l'arborescence d'un objet message est plutôt simple avec la méthode `Message.walk`. Le module `email.iterators` fournit des fonctionnalités d'itérations de plus haut niveau sur les arbres d'objets messages.

`email.iterators.body_line_iterator(msg, decode=False)`

Cette fonction permet d'itérer sur tous les contenus de tous les éléments de *msg*, en retournant les contenus sous forme de chaînes de caractères ligne par ligne. Il saute les entêtes des sous éléments, et tous les sous éléments dont le contenu n'est pas une chaîne de caractères Python. C'est en quelque sorte équivalent à une lecture plate d'une représentation textuelle du message à partir d'un fichier en utilisant `readline()`, et en sautant toutes les entêtes intermédiaires.

Le paramètre optionnel *decode* est transmis à la méthode `Message.get_payload`.

`email.iterators.typed_subpart_iterator(msg, maintype='text', subtype=None)`

Cette fonction permet d'itérer sur tous les sous éléments de *msg*, en retournant seulement les sous éléments qui correspondent au type MIME spécifié par *maintype* et *subtype*.

Notez que le paramètre *subtype* est optionnel; s'il n'est pas présent, alors le type MIME du sous élément est seulement composé du type principal. *maintype* est également optionnel; sa valeur par défaut est *text*.

En conséquence, par défaut, `typed_subpart_iterator()` retourne chaque sous élément qui a un type MIME de type *text/**.

La fonction suivante a été ajoutée en tant qu'un outil de débogage. Elle *ne devrait pas* être considérée comme une interface publique supportée pour ce paquet.

`email.iterators._structure(msg, fp=None, level=0, include_default=False)`

Affiche une représentation indentée des types de contenu de la structure de l'objet message. Par exemple :

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
  text/plain
  text/plain
  multipart/digest
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
  text/plain
```

Le paramètre optionnel *fp* est un objet fichier-compatible dans lequel on peut écrire le flux de sortie. Il doit être approprié pour la fonction de Python `print()`. *level* est utilisé en interne. *include_default*, si vrai, affiche aussi le type par défaut.

Voir aussi :

Module `smtplib` Client SMTP (*Simple Mail Transport Protocol*)

Module `poplib` Client POP (*Post Office Protocol*)

Module `imaplib` Client IMAP (*Internet Message Access Protocol*)

Module `nntplib` Client NNTP (*Net News Transport Protocol*)

Module `mailbox` Outils pour créer, lire et gérer des messages regroupés sur disque en utilisant des formats standards variés.

Module `smtpd` Cadriciel pour serveur SMTP (principalement utile pour tester)

19.2 json — Encodage et décodage JSON

Code source : [Lib/json/__init__.py](#)

JSON (JavaScript Object Notation), spécifié par la **RFC 7159** (qui rend la **RFC 4627** obsolète) et par le standard ECMA-404, est une interface légère d'échange de données inspirée par la syntaxe des objets littéraux JavaScript (bien que ce ne soit pas un sous-ensemble strict de Javascript¹).

json expose une API familière aux utilisateurs des modules de la bibliothèque standard *marshal* et *pickle*.

Encodage d'objets Python basiques :

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\\bar\""))
\"foo\\bar\"
>>> print(json.dumps('\\u1234'))
```

(suite sur la page suivante)

1. Comme noté dans l'errata de la RFC 7159, JSON autorise les caractères littéraux U+2028 (*LINE SEPARATOR*) et U+2029 (*PARAGRAPH SEPARATOR*) dans les chaînes de caractères, alors que Javascript (selon le standard ECMAScript édition 5.1) ne le permet pas.

(suite de la page précédente)

```

"\u1234"
>>> print(json.dumps('\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'

```

Encodage compact :

```

>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'

```

Affichage élégant :

```

>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}

```

Décodage JSON :

```

>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\"foo\\bar\"')
'foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']

```

Spécialisation du décodage JSON pour un objet :

```

>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')

```

Étendre la classe *JSONEncoder* :

```

>>> import json
>>> class ComplexEncoder(json.JSONEncoder):

```

(suite sur la page suivante)

(suite de la page précédente)

```

...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', ', '1.0', ', ', ']']

```

Utiliser `json.tool` depuis le *shell* pour valider et afficher élégamment :

```

$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)

```

Voir *Interface en ligne de commande* pour une documentation détaillée.

Note : JSON est un sous-ensemble de [YAML 1.2](#). Le JSON produit par les paramètres par défaut de ce module (en particulier, la valeur par défaut de `separators`) est aussi un sous ensemble de [YAML 1.0](#) et [1.1](#). Ce module peut alors aussi être utilisé comme sérialiseur [YAML](#).

19.2.1 Utilisation basique

`json.dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)`

Séréalise `obj` comme un flux JSON formaté vers `fp` (un *file-like object* gérant `.write()`) utilisant cette *table de conversion*.

Si `skipkeys` est vrai (faux par défaut), alors les clefs de dictionnaires qui ne sont pas de types basiques (`str`, `int`, `float`, `bool`, `None`) seront ignorées, elles provoquent normalement la levée d'une *TypeError*.

Le module `json` produit toujours des objets `str`, et non des objets `bytes`. `fp.write()` doit ainsi supporter un objet `str` en entrée.

Si `ensure_ascii` est vrai (par défaut), il est garanti que les caractères non ASCII soient tous échappés sur la sortie.

Si `ensure_ascii` est faux, ces caractères seront écrits comme tels.

Si `check_circular` est faux (vrai par défaut), la vérification des références circulaires pour les conteneurs sera ignorée, et une référence circulaire résultera en une *OverflowError* (ou pire).

Si `allow_nan` est faux (vrai par défaut), une *ValueError* sera levée lors de la sérialisation de valeurs `float` extérieures aux bornes (`nan`, `inf`, `-inf`), en respect strict de la spécification JSON. Si `allow_nan` est vrai, leurs équivalents JavaScript (`NaN`, `Infinity`, `-Infinity`) seront utilisés.

Si `indent` est un nombre entier positif ou une chaîne de caractères, les éléments de tableaux et les membres d'objets JSON seront affichés élégamment avec ce niveau d'indentation. Un niveau d'indentation de 0, négatif, ou "" n'insérera que des retours à la ligne. `None` (la valeur par défaut) choisit la représentation la plus compacte. Utiliser un entier positif comme indentation indente d'autant d'espaces par niveau. Si `indent` est une chaîne (telle que "\t"), cette chaîne est utilisée pour indenter à chaque niveau.

Modifié dans la version 3.2 : Autorise les chaînes en plus des nombres entiers pour `indent`.

Si spécifié, *separators* doit être un *tuple* (*item_separator*, *key_separator*). Sa valeur par défaut est (' ', ': ') si *indent* est *None*, et (' ', ': ') autrement. Pour obtenir la représentation JSON la plus compacte possible, vous devriez spécifier ('', ':') pour éliminer les espaces.

Modifié dans la version 3.4 : Utilise (' ', ': ') par défaut si *indent* n'est pas *None*.

Si spécifié, *default* doit être une fonction qui sera appelée pour les objets qui ne peuvent être sérialisés autrement. Elle doit renvoyer une représentation de l'objet sérialisable en JSON ou lever une *TypeError*. Si non spécifié, une *TypeError* sera levée pour les types non sérialisables.

Si *sort_keys* est vrai (faux par défaut), les dictionnaires seront retranscrits triés selon leurs clés.

Pour utiliser une sous-classe *JSONEncoder* personnalisée (p. ex. une qui redéfinit la méthode *default()* pour sérialiser des types additionnels), spécifiez-la avec le paramètre nommé *cls*; autrement, *JSONEncoder* est utilisée.

Modifié dans la version 3.6 : Tous les paramètres optionnels sont maintenant des *keyword-only*.

Note : À l'inverse de *pickle* et *marshal*, JSON n'est pas un protocole par trames, donc essayer de sérialiser de multiples objets par des appels répétés à *dump()* en utilisant le même *fp* resultera en un fichier JSON invalide.

`json.dumps(obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)`
Sérialise *obj* vers un JSON formaté *str*, en utilisant cette *table de conversion*. Les arguments ont la même signification que ceux de *dump()*.

Note : Les clés dans les couples JSON clé/valeur sont toujours de type *str*. Quand un dictionnaire est converti en JSON, toutes les clés du dictionnaire sont transformées en chaînes de caractères. Ce qui fait que si un dictionnaire est converti en JSON et reconverti en dictionnaire, le résultat peut ne pas être égal à l'original. Ainsi, `loads(dumps(x)) != x` si *x* contient des clés qui ne sont pas des chaînes.

`json.load(fp, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)`
Déserialise *fp* (un *text file* ou un *binary file* supportant *.read()* et contenant un document JSON) vers un objet Python en utilisant cette *table de conversion*.

object_hook est une fonction optionnelle qui sera appelée avec le résultat de chaque objet littéral décodé (chaque *dict*). La valeur de retour de *object_hook* sera utilisée à la place du *dict*. Cette fonctionnalité peut être utilisée pour implémenter des décodeurs personnalisés (p. ex. les *class hinting* de *JSON-RPC*).

object_pairs_hook est une fonction optionnelle qui sera appelé pour chaque objet littéral décodé, avec une liste ordonnée de couples. La valeur de retour de *object_pairs_hook* sera utilisée à la place du *dict*. Cette fonctionnalité peut être utilisée pour implémenter des décodeurs personnalisés qui s'appuient sur l'ordre dans lequel les couples clé/valeur sont décodés (par exemple, *collections.OrderedDict()* mémorisera l'ordre d'insertion). *object_pairs_hook* prend la priorité sur *object_hook*, si cette dernière est aussi définie.

Modifié dans la version 3.1 : Ajout du support de *object_pairs_hook*.

parse_float, si spécifiée, sera appelée pour chaque nombre réel JSON à décoder sous forme d'une chaîne de caractères. Par défaut, elle est équivalente à `float(num_str)`. Cela peut servir à utiliser un autre type de données ou un autre analyseur pour les nombres réels JSON (p. ex. *decimal.Decimal*).

parse_int, si spécifiée, sera appelée pour chaque nombre entier JSON à décoder sous forme d'une chaîne de caractères. Par défaut, elle est équivalente à `int(num_str)`. Cela peut servir à utiliser un autre type de données ou un autre analyseur pour les nombres entiers JSON (p. ex. *float*).

parse_constant, si spécifiée, sera appelée avec l'une des chaînes de caractères suivantes : *'-Infinity'*, *'Infinity'* ou *'NaN'*. Cela peut servir à lever une exception si des nombres JSON invalides sont rencontrés.

Modifié dans la version 3.1 : *parse_constant* n'est plus appelée pour *null*, *true* ou *false*.

Pour utiliser une sous-classe *JSONDecoder* personnalisée, spécifiez-la avec l'argument nommé *cls*; autrement, *JSONDecoder* est utilisée. Les arguments nommés additionnels seront passés au constructeur de cette classe.

Si les données à désérialiser ne sont pas un document JSON valide, une `JSONDecodeError` sera levée.

Modifié dans la version 3.6 : Tous les paramètres optionnels sont maintenant des *keyword-only*.

Modifié dans la version 3.6 : `fp` peut maintenant être un `binary file`. Son encodage doit être UTF-8, UTF-16 ou UTF-32.

```
json.loads(s, *, encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None,
           parse_constant=None, object_pairs_hook=None, **kw)
```

Désérialise `s` (une instance de `str`, `bytes` ou `bytearray` contenant un document JSON) vers un objet Python en utilisant cette *table de conversion*.

Les autres arguments ont la même signification que pour `load()`, à l'exception d'`encoding` qui est ignoré et obsolète.

Si les données à désérialiser ne sont pas un document JSON valide, une `JSONDecodeError` sera levée.

Modifié dans la version 3.6 : `s` peut maintenant être de type `bytes` ou `bytearray`. L'encodage d'entrée doit être UTF-8, UTF-16 ou UTF-32.

19.2.2 Encodeurs et décodeurs

```
class json.JSONDecoder(*, object_hook=None, parse_float=None, parse_int=None,
                       parse_constant=None, strict=True, object_pairs_hook=None)
```

Décodeur simple JSON.

Applique par défaut les conversions suivantes en décodant :

JSON	Python
objet	<i>dict</i>
array	<i>list</i>
string	<i>str</i>
number (nombre entier)	<i>int</i>
number (nombre réel)	<i>float</i>
true	<i>True</i>
false	<i>False</i>
null	<i>None</i>

Les valeurs `NaN`, `Infinity` et `-Infinity` sont aussi comprises comme leurs valeurs `float` correspondantes, bien que ne faisant pas partie de la spécification JSON.

`object_hook`, si spécifiée, sera appelée avec le résultat de chaque objet JSON décodé et sa valeur de retour sera utilisée à la place du *dict* donné. Cela peut être utilisé pour apporter des désérialisations personnalisées (p. ex. pour supporter les *class hinting* de JSON-RPC).

`object_pairs_hook`, si spécifiée, sera appelée avec le résultat de chaque objet JSON décodé avec une liste ordonnée de couples. Sa valeur de retour sera utilisée à la place du *dict*. Cette fonctionnalité peut être utilisée pour implémenter des décodeurs personnalisés se basant sur l'ordre dans lequel les couples clé/valeur sont décodés (par exemple, `collections.OrderedDict()` mémorisera l'ordre d'insertion). `object_pairs_hook` prend la priorité sur `object_hook`, si cette dernière est aussi définie.

Modifié dans la version 3.1 : Ajout du support de `object_pairs_hook`.

`parse_float`, si spécifiée, sera appelée pour chaque nombre réel JSON à décoder sous forme d'une chaîne de caractères. Par défaut, elle est équivalente à `float(num_str)`. Cela peut servir à utiliser un autre type de données ou un autre analyseur pour les nombres réels JSON (p. ex. `decimal.Decimal`).

`parse_int`, si spécifiée, sera appelée pour chaque nombre entier JSON à décoder sous forme d'une chaîne de caractères. Par défaut, elle est équivalente à `int(num_str)`. Cela peut servir à utiliser un autre type de données ou un autre analyseur pour les nombres entiers JSON (p. ex. `float`).

`parse_constant`, si spécifiée, sera appelée avec l'une des chaînes de caractères suivantes : `'-Infinity'`, `'Infinity'` ou `'NaN'`. Cela peut servir à lever une exception si des nombres JSON invalides sont rencontrés.

Si *strict* est faux (*True* par défaut), alors les caractères de contrôle seront autorisés à l'intérieur des chaînes. Les caractères de contrôle dans ce contexte sont ceux dont les codes sont dans l'intervalle 0–31, incluant `'\t'` (tabulation), `'\n'`, `'\r'` et `'\0'`.

Si les données à désérialiser ne sont pas un document JSON valide, une *JSONDecodeError* sera levée.

Modifié dans la version 3.6 : Tous les paramètres sont maintenant des *keyword-only*.

decode (*s*)

Renvoie la représentation Python de *s* (une instance *str* contenant un document JSON).

Une *JSONDecodeError* sera levée si le document JSON donné n'est pas valide.

raw_decode (*s*)

Décode en document JSON depuis *s* (une instance *str* débutant par un document JSON) et renvoie un *tuple* de 2 éléments contenant la représentation Python de l'objet et l'index dans *s* où le document se terminait.

Elle peut être utilisée pour décoder un document JSON depuis une chaîne qui peut contenir des données supplémentaires à la fin.

class `json.JSONEncoder` (*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Encodeur JSON extensible pour les structures de données Python.

Supporte par défaut les objets et types suivants :

Python	JSON
<i>dict</i>	objet
<i>list, tuple</i>	<i>array</i>
<i>str</i>	<i>string</i>
<i>int, float</i> , et <i>Enums</i> dérivées d' <i>int</i> ou de <i>float</i>	<i>number</i>
<i>True</i>	<i>true</i>
<i>False</i>	<i>false</i>
<i>None</i>	<i>null</i>

Modifié dans la version 3.4 : Ajout du support des classes *Enum* dérivées d'*int* ou de *float*.

Pour l'étendre afin de reconnaître d'autres types d'objets, il suffit d'en créer une sous-classe et d'implémenter une nouvelle méthode *default()* qui renverrait si possible un objet sérialisable pour *o*, ou ferait appel à l'implémentation de la classe mère (qui lèverait une *TypeError*).

Si *skipkeys* est faux (par défaut), une *TypeError* sera levée lors de l'encodage de clés autres que des *str*, des *int*, des *float* ou *None*. Si *skipkeys* est vrai, ces éléments sont simplement ignorés.

Si *ensure_ascii* est vrai (par défaut), il est garanti que les caractères non ASCII soient tous échappés sur la sortie. Si *ensure_ascii* est faux, ces caractères seront écrits comme tels.

Si *check_circular* est vrai (par défaut), une vérification aura lieu sur les listes, dictionnaires et objets personnalisés, afin de détecter les références circulaires et éviter les récursions infinies (qui causeraient une *OverflowError*). Autrement, la vérification n'a pas lieu.

Si *allow_nan* est vrai (par défaut), alors *NaN*, *Infinity* et *-Infinity* seront encodés comme tels. Ce comportement ne respecte pas la spécification JSON, mais est cohérent avec la majorité des encodeurs/décodeurs JavaScript. Autrement, une *ValueError* sera levée pour de telles valeurs.

Si *sort_keys* est vrai (*False* par défaut), alors les dictionnaires seront triés par clés en sortie ; cela est utile lors de tests de régression pour pouvoir comparer les sérialisations JSON au jour le jour.

Si *indent* est un nombre entier positif ou une chaîne de caractères, les éléments de tableaux et les membres d'objets JSON seront affichés élégamment avec ce niveau d'indentation. Un niveau d'indentation de 0, négatif, ou "" n'insérera que des retours à la ligne. *None* (la valeur par défaut) choisit la représentation la plus compacte. Utiliser un entier positif comme indentation indente d'autant d'espaces par niveau. Si *indent* est une chaîne (telle que "\t"), cette chaîne est utilisée pour indenter à chaque niveau.

Modifié dans la version 3.2 : Autorise les chaînes en plus des nombres entiers pour *indent*.

Si spécifié, *separators* doit être un *tuple* (*item_separator*, *key_separator*). Sa valeur par défaut est (' ', ': ') si *indent* est *None*, et ('', ': ') autrement. Pour obtenir la représentation JSON la plus compacte possible, vous devriez spécifier ('', ': ') pour éliminer les espacements.

Modifié dans la version 3.4 : Utilise `(' ', ' ', ' : ')` par défaut si *indent* n'est pas `None`.

Si spécifié, *default* doit être une fonction qui sera appelée pour les objets qui ne peuvent être sérialisés autrement. Elle doit renvoyer une représentation de l'objet sérialisable en JSON ou lever une `TypeError`. Si non spécifié, une `TypeError` sera levée pour les types non sérialisables.

Modifié dans la version 3.6 : Tous les paramètres sont maintenant des *keyword-only*.

default (*o*)

Implémentez cette méthode dans une sous-classe afin qu'elle renvoie un objet sérialisable pour *o*, ou appelle l'implémentation de base (qui lèvera une `TypeError`).

Par exemple, pour supporter des itérateurs arbitraires, vous pourriez implémenter *default* comme cela :

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return json.JSONEncoder.default(self, o)
```

encode (*o*)

Renvoie une chaîne JSON représentant la structure de données Python *o*. Par exemple :

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode (*o*)

Encode l'objet *o* donné, et produit chaque chaîne représentant l'objet selon disponibilité. Par exemple :

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

19.2.3 Exceptions

exception `json.JSONDecodeError` (*msg, doc, pos*)

Sous-classe de `ValueError` avec les attributs additionnels suivants :

msg

Le message d'erreur non formaté.

doc

Le document JSON actuellement traité.

pos

L'index de *doc* à partir duquel l'analyse a échoué.

lineno

La ligne correspondant à *pos*.

colno

La colonne correspondant à *pos*.

Nouveau dans la version 3.5.

19.2.4 Conformité au standard et Interopérabilité

Le format JSON est spécifié par la [RFC 7159](#) et le standard [ECMA-404](#). Cette section détaille la conformité à la RFC au niveau du module. Pour faire simple, les sous-classes de `JSONEncoder` et `JSONDecoder`, et les paramètres autres que ceux explicitement mentionnés ne sont pas considérés.

Ce module ne se conforme pas strictement à la RFC, implémentant quelques extensions qui sont valides en JavaScript mais pas en JSON. En particulier :

- Les nombres infinis et *NaN* sont acceptés et retranscrits ;
- Les noms répétés au sein d'un objet sont acceptés, seule la valeur du dernier couple nom/valeur sera utilisée.

Comme la RFC permet aux analyseurs conformes d'accepter des textes en entrée non conformes, le déserialiseur de ce module avec ses paramètres par défaut est techniquement conforme à la RFC.

Encodage des caractères

La RFC requiert que le JSON soit représenté en utilisant l'encodage UTF-8, UTF-16 ou UTF-32, avec UTF-8 recommandé par défaut pour une interopérabilité maximale.

Comme cela est permis par la RFC, bien que non requis, le sérialiseur du module active `ensure_ascii=True` par défaut, échappant ainsi la sortie de façon à ce que les chaînes résultants ne contiennent que des caractères ASCII.

Outre le paramètre `ensure_ascii`, les conversions entre objets Python et chaînes *Unicode* de ce module sont strictement définies, et ne rencontrent donc pas directement le problème de l'encodage des caractères.

La RFC interdit d'ajouter un *byte* marqueur d'ordre (BOM) au début du texte JSON, et le sérialiseur de ce module n'ajoute pas de tel BOM. La RFC permet, mais ne requiert pas, que les déserialiseurs JSON ignorent ces BOM. Le déserialiseur de ce module lève une `ValueError` quand un BOM est présent au début du fichier.

La RFC n'interdit pas explicitement les chaînes JSON contenant des séquences de *bytes* ne correspondant à aucun caractère Unicode valide (p. ex. les *surrogates* UTF-16 sans correspondance), mais précise que cela peut causer des problèmes d'interopérabilité. Par défaut, ce module accepte et retranscrit (quand présents dans la *str* originale) les *code points* de telles séquences.

Valeurs numériques infinies et NaN

La RFC ne permet pas la représentation des nombres infinis ou des *NaN*. Néanmoins, par défaut, ce module accepte et retranscrit `Infinity`, `-Infinity` et `NaN` comme s'ils étaient des valeurs numériques littérales JSON valides :

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

Dans le sérialiseur, le paramètre `allow_nan` peut être utilisé pour altérer ce comportement. Dans le déserialiseur, le paramètre `parse_constant` peut être utilisé pour altérer ce comportement.

Noms répétés au sein d'un objet

La RFC spécifie que les noms au sein d'un objet JSON doivent être uniques, mais ne décrit pas comment les noms répétés doivent être gérés. Par défaut, ce module ne lève pas d'exception ; à la place, il ignore tous les couples nom/valeur sauf le dernier pour un nom donné :

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

Le paramètre *object_pairs_hook* peut être utilisé pour altérer ce comportement.

Valeurs de plus haut niveau autres qu'objets ou tableaux

L'ancienne version de JSON spécifiée par l'obsolète [RFC 4627](#) demandait à ce que la valeur de plus haut niveau du texte JSON soit un objet ou un tableau JSON (*dict* ou *list* Python), et ne soit pas *null*, un nombre, ou une chaîne de caractères. La [RFC 7159](#) a supprimé cette restriction, jamais implémentée par ce module, que ce soit dans le sérialiseur ou le désérialiseur.

Cependant, pour une interopérabilité maximale, vous pourriez volontairement souhaiter adhérer à cette restriction par vous-même.

Limitations de l'implémentation

Certaines implémentations de désérialiseurs JSON peuvent avoir des limites sur :

- la taille des textes JSON acceptés ;
- le niveau maximum d'objets et tableaux JSON imbriqués ;
- l'intervalle et la précision des nombres JSON ;
- le contenu et la longueur maximale des chaînes JSON.

Ce module n'impose pas de telles limites si ce n'est celles inhérentes aux types de données Python ou à l'interpréteur.

Lors d'une sérialisation JSON, faites attention à ces limitations dans les applications qui utiliseraient votre JSON. En particulier, il est commun pour les nombres JSON d'être désérialisés vers des nombres IEEE 754 à précision double, et donc sujets à l'intervalle et aux limitations sur la précision de cette représentation. Cela est d'autant plus important lors de la sérialisation de valeurs *int* Python de forte magnitude, ou d'instances de types numériques « exotiques » comme *decimal.Decimal*.

19.2.5 Interface en ligne de commande

Code source : [Lib/json/tool.py](#)

Le module *json.tool* fournit une simple interface en ligne de commande pour valider et réécrire élégamment des objets JSON.

Si les arguments optionnels *infile* et *outfile* ne sont pas spécifiés, *sys.stdin* et *sys.stdout* seront utilisés respectivement :

```
$ echo '{"json": "obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```


Modifié dans la version 3.5 : La sortie conserve maintenant l'ordre des données de l'entrée. Utilisez l'option `--sort-keys` pour sortir des dictionnaires triés alphabétiquement par clés.

Options de la ligne de commande

infile

Le fichier JSON à valider ou réécrire élégamment :

```
$ python -m json.tool mp_films.json
[
  {
    "title": "And Now for Something Completely Different",
    "year": 1971
  },
  {
    "title": "Monty Python and the Holy Grail",
    "year": 1975
  }
]
```

Si *infile* n'est pas spécifié, lit le document depuis `sys.stdin`.

outfile

Écrit la sortie générée par *infile* vers le fichier *outfile* donné. Autrement, écrit sur `sys.stdout`.

--sort-keys

Trie alphabétiquement les dictionnaires par clés.

Nouveau dans la version 3.5.

-h, --help

Affiche le message d'aide.

Notes

19.3 mailcap — Manipulation de fichiers Mailcap

Code source : [Lib/mailcap.py](#)

Les fichiers *mailcap* sont utilisés pour configurer la façon dont les applications compatibles avec MIME, comme les clients mails ou les navigateurs, réagissent aux différents fichiers de types MIME. (Le nom *mailcap* est une contraction de l'expression « *mail capability* ».) Par exemple, un fichier *mailcap* peut contenir une ligne de type `video/mpeg; xmpeg %s`. Ensuite, si l'utilisateur récupère un message mail ou un document web avec un type MIME `video/mpeg`, `%s` est remplacé par un nom de fichier (généralement celui d'un fichier temporaire) et le programme **xmpeg** peut automatiquement débiter la lecture de ce dernier.

Le format *mailcap* est documenté dans la **RFC 1524** « A User Agent Configuration Mechanism For Multimedia Mail Format Information », mais n'est pas un standard Internet. Cependant, la plupart des systèmes Unix savent gérer les fichiers *mailcap*.

`mailcap.findmatch(caps, MIMEtype, key='view', filename='/dev/null', plist=[])`

Renvoie un tuple à deux éléments ; le premier élément est une chaîne de caractères (string) contenant la ligne de commande à exécuter (qui peut être passée à `os.system()`), et le second élément est l'entrée *mailcap* pour un type de MIME donné. Si le type MIME n'est pas identifié, `(None, None)` est renvoyé.

key est le nom de champ souhaité, qui représente le type d'action à exécuter ; la valeur par défaut est 'view', puisque dans la majorité des cas le besoin consiste juste à lire le corps (body) de la donnée de type MIME. Les autres valeurs possibles peuvent être 'compose' et 'edit', si le besoin consiste à créer un nouveau corps de données (body) ou modifier celui existant. Voir la [RFC 1524](#) pour une liste complète des champs.

filename est le nom de fichier à remplacer pour %s en ligne de commande ; la valeur par défaut est '/dev/null' qui n'est certainement pas celle que vous attendez. Donc la plupart du temps, le nom de fichier doit être indiqué.

plist peut être une liste contenant des noms de paramètres ; la valeur par défaut est une simple liste vide. Chaque entrée dans la liste doit être une chaîne de caractères contenant le nom du paramètre, un signe égal ('='), ainsi que la valeur du paramètre. Les entrées *mailcap* peuvent contenir des noms de paramètres tels que % {foo}, remplacé par la valeur du paramètre nommé *foo*. Par exemple, si la ligne de commande `showpartial % {id} % {number} % {total}` est un fichier *mailcap*, et *plist* configuré à ['id=1', 'number=2', 'total=3'], la ligne de commande qui en résulte est 'showpartial 1 2 3'.

Dans un fichier *mailcap*, le champ « test » peut être renseigné de façon optionnelle afin de tester certaines conditions externes (comme l'architecture machine, ou le gestionnaire de fenêtre utilisé) afin de déterminer si la ligne *mailcap* est pertinente ou non. *findmatch()* vérifie automatiquement ces conditions et ignore l'entrée si la vérification échoue.

`mailcap.getcaps()`

Renvoie un dictionnaire qui associe les types MIME à une liste d'entrées de fichier *mailcap*. Ce dictionnaire doit être transmis à la fonction *findmatch()*. Une entrée est enregistrée en tant qu'une liste de dictionnaires, mais il n'est pas nécessaire de connaître les détails de cette représentation.

L'information provient de tous les fichiers *mailcap* trouvés dans le système. Les configurations réalisées dans le fichier *mailcap* du répertoire utilisateur `$HOME/.mailcap` outrepassent les configurations systèmes des fichiers *mailcap* `/etc/mailcap`, `/usr/etc/mailcap`, et `/usr/local/etc/mailcap`.

Un exemple d'utilisation :

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

19.4 mailbox — Manipuler les boîtes de courriels dans différents formats

Code source : [Lib/mailbox.py](#)

Ce module définit deux classes, *Mailbox* et *Message*, pour accéder et manipuler les boîtes de courriel sur le disque et les messages qu'elles contiennent. *Mailbox* offre une interface ressemblant aux dictionnaires avec des clés et des messages. La classe *Message* étend le module *email.message* de la classe *Message* avec un état et un comportement spécifiques à son format. Les formats de boîtes de courriel gérés sont *Maildir*, *mbox*, *MH*, *Babyl* et *MMDf*.

Voir aussi :

Module *email* Représente et manipule des messages.

19.4.1 Objets Mailbox

class mailbox.Mailbox

Une boîte mail, qui peut être inspectée et modifiée.

La classe *Mailbox* définit une interface et n'est pas destinée à être instanciée. Les sous-classes de format spécifique doivent plutôt hériter de *Mailbox* et votre code doit instancier une sous-classe particulière.

L'interface *Mailbox* est compatible avec celle des dictionnaires, avec de courtes clés correspondant aux messages. Les clés sont générées par l'instance *Mailbox* avec laquelle elles sont utilisées et n'ont de sens que pour cette instance *Mailbox*. Une clé continue d'identifier un message même si le message correspondant est modifié ou remplacé par un autre message.

Les messages peuvent être ajoutés à une instance *Mailbox* en utilisant la méthode *add()* (comme pour les ensembles), et supprimés en utilisant soit l'instruction *del* soit les méthodes *remove()* et *discard()* (comme pour les ensembles).

La sémantique de l'interface *Mailbox* diffère de la sémantique des dictionnaires sur plusieurs aspects. À chaque fois qu'un message est demandé, une nouvelle représentation (généralement une instance *Message*) est générée en se basant sur l'état actuel de la boîte mail. De la même manière, lorsqu'un message est ajouté à l'instance *Mailbox*, le contenu de la représentation du message donné est copié. En aucun cas une référence vers la représentation du message n'est gardée par l'instance *Mailbox*.

L'itérateur par défaut de *Mailbox* itère sur les représentations des messages et pas sur les clés (comme le fait par défaut l'itérateur des dictionnaires). De plus, les modifications sur une boîte mail durant l'itération sont sûres et clairement définies. Les messages ajoutés à la boîte mail après la création d'un itérateur ne sont pas vus par l'itérateur. Les messages supprimés de la boîte mail avant que l'itérateur les traite seront ignorés silencieusement. Toutefois, utiliser une clé depuis un itérateur peut aboutir à une exception *KeyError* si le message correspondant est supprimé par la suite.

Avertissement : Soyez très prudent lorsque vous éditez des boîtes mail qui peuvent être modifiées par d'autres processus. Le format de boîte mail le plus sûr à utiliser pour ces tâches est *Maildir*, essayez d'éviter les formats à fichier unique tels que *mbx* afin d'empêcher les écritures concurrentes. Si vous modifiez une boîte mail, vous devez la verrouiller en appelant les méthodes *lock()* et *unlock()* avant de lire les messages dans le fichier ou d'y appliquer des changements en y ajoutant ou supprimant des messages. Ne pas verrouiller la boîte mail vous fait prendre le risque de perdre des messages ou de corrompre la boîte mail entière.

Les instances *Mailbox* contiennent les méthodes suivantes :

add (*message*)

Ajoute *message* à la boîte mail et renvoie la clé qui lui a été assigné.

Le paramètre *message* peut être une instance *Message*, une instance *email.message.Message*, une chaîne de caractères, une séquence d'octets ou un objet fichier-compatible (qui doit être ouvert en mode binaire). Si *message* est une instance de la sous-classe *Message* au format correspondant (par exemple s'il s'agit d'une instance *mbxMessage* et d'une instance *mbx*), les informations spécifiques à son format sont utilisées. Sinon, des valeurs par défaut raisonnables pour son format sont utilisées.

Modifié dans la version 3.2 : Ajout de la gestion des messages binaires.

remove (*key*)

__delitem__ (*key*)

discard (*key*)

Supprime le message correspondant à *key* dans la boîte mail.

Si ce message n'existe pas, une exception *KeyError* est levée si la méthode a été appelée en tant que *remove()* ou *__delitem__()* mais aucune exception n'est levée si la méthode a été appelée en tant que *discard()*. Vous préférerez sûrement le comportement de *discard()* si le format de boîte mail sous-jacent accepte la modification concurrente par les autres processus.

__setitem__ (*key*, *message*)

Remplace le message correspondant à *key* par *message*. Lève une exception *KeyError* s'il n'y a pas déjà de message correspondant à *key*.

Comme pour `add()`, le paramètre `message` peut être une instance `Message`, une instance `email.message.Message`, une chaîne de caractères, une chaîne d'octets ou un objet fichier-compatible (qui doit être ouvert en mode binaire). Si `message` est une instance de la sous-classe `Message` au format correspondant (par exemple s'il s'agit d'une instance `mbxMessage` et d'une instance `mbx`), les informations spécifiques à son format sont utilisées. Sinon, les informations spécifiques au format du message qui correspond à `key` ne sont modifiées.

iterkeys()

keys()

Renvoie un itérateur sur toutes les clés s'il est appelé en tant que `iterkeys()` ou renvoie une liste de clés s'il est appelé en tant que `keys()`.

intervalues()

__iter__()

values()

Renvoie un itérateur sur les représentations de tous les messages s'il est appelé en tant que `intervalues()` ou `__iter__()` et renvoie une liste de ces représentations s'il est appelé en tant que `values()`. Les messages sont représentés en tant qu'instances de la sous-classe `Message` au format correspondant à moins qu'une fabrique de messages personnalisée soit spécifiée lorsque l'instance `Mailbox` a été initialisée.

Note : The behavior of `__iter__()` is unlike that of dictionaries, which iterate over keys.

iteritems()

items()

Return an iterator over (`key`, `message`) pairs, where `key` is a key and `message` is a message representation, if called as `iteritems()` or return a list of such pairs if called as `items()`. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get(key, default=None)

__getitem__(key)

Return a representation of the message corresponding to `key`. If no such message exists, `default` is returned if the method was called as `get()` and a `KeyError` exception is raised if the method was called as `__getitem__()`. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get_message(key)

Return a representation of the message corresponding to `key` as an instance of the appropriate format-specific `Message` subclass, or raise a `KeyError` exception if no such message exists.

get_bytes(key)

Return a byte representation of the message corresponding to `key`, or raise a `KeyError` exception if no such message exists.

Nouveau dans la version 3.2.

get_string(key)

Return a string representation of the message corresponding to `key`, or raise a `KeyError` exception if no such message exists. The message is processed through `email.message.Message` to convert it to a 7bit clean representation.

get_file(key)

Return a file-like representation of the message corresponding to `key`, or raise a `KeyError` exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

Modifié dans la version 3.2 : The file object really is a binary file; previously it was incorrectly returned in text mode. Also, the file-like object now supports the context management protocol : you can use a `with` statement to automatically close it.

Note : Unlike other representations of messages, file-like representations are not necessarily independent of the `Mailbox` instance that created them or of the underlying mailbox. More specific documentation is

provided by each subclass.

__contains__ (*key*)

Return True if *key* corresponds to a message, False otherwise.

__len__ ()

Return a count of messages in the mailbox.

clear ()

Delete all messages from the mailbox.

pop (*key*, *default=None*)

Return a representation of the message corresponding to *key* and delete the message. If no such message exists, return *default*. The message is represented as an instance of the appropriate format-specific *Message* subclass unless a custom message factory was specified when the *Mailbox* instance was initialized.

popitem ()

Return an arbitrary (*key*, *message*) pair, where *key* is a key and *message* is a message representation, and delete the corresponding message. If the mailbox is empty, raise a *KeyError* exception. The message is represented as an instance of the appropriate format-specific *Message* subclass unless a custom message factory was specified when the *Mailbox* instance was initialized.

update (*arg*)

Parameter *arg* should be a *key*-to-*message* mapping or an iterable of (*key*, *message*) pairs. Updates the mailbox so that, for each given *key* and *message*, the message corresponding to *key* is set to *message* as if by using *__setitem__* (). As with *__setitem__* (), each *key* must already correspond to a message in the mailbox or else a *KeyError* exception will be raised, so in general it is incorrect for *arg* to be a *Mailbox* instance.

Note : Unlike with dictionaries, keyword arguments are not supported.

flush ()

Write any pending changes to the filesystem. For some *Mailbox* subclasses, changes are always written immediately and *flush* () does nothing, but you should still make a habit of calling this method.

lock ()

Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An *ExternalClashError* is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

unlock ()

Release the lock on the mailbox, if any.

close ()

Flush the mailbox, unlock it if necessary, and close any open files. For some *Mailbox* subclasses, this method does nothing.

Maildir

class mailbox.**Maildir** (*dirname*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in Maildir format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is None, *MaildirMessage* is used as the default message representation. If *create* is True, the mailbox is created if it does not exist.

It is for historical reasons that *dirname* is named as such rather than *path*.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely : `tmp`, `new`, and `cur`. Messages are created momentarily in the `tmp` subdirectory and then moved to the `new` subdirectory to finalize delivery. A mail user agent may subsequently move the message to the `cur` subdirectory and store information about the state of the message in a special « info » section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if `'.'` is the first character in its name. Folder names are represented by *Maildir* without the leading `'.'`. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using `'.'` to delimit levels, e.g., « Archived.2005.07 ».

Note : The Maildir specification requires the use of a colon (`:`) in certain message file names. However, some operating systems do not permit this character in file names, If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (`!`) is a popular choice. For example :

```
import mailbox
mailbox.Maildir.colon = '!'
```

The `colon` attribute may also be set on a per-instance basis.

Maildir instances have all of the methods of *Mailbox* in addition to the following :

list_folders()

Return a list of the names of all folders.

get_folder(folder)

Return a *Maildir* instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(folder)

Create a folder whose name is *folder* and return a *Maildir* instance representing it.

remove_folder(folder)

Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

clean()

Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

Some *Mailbox* methods implemented by *Maildir* deserve special remarks :

add(message)

__setitem__(key, message)

update(arg)

Avertissement : These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()

All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()

unlock()

Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()

Maildir instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file (*key*)

Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

Voir aussi :

maildir man page from gmail The original specification of the format.

Using maildir format Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on « info » semantics.

maildir man page from Courier Another specification of the format. Describes a common extension for supporting folders.

mbx

class mailbox.**mbx** (*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in mbox format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is *None*, *mbxMessage* is used as the default message representation. If *create* is *True*, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are « From ».

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, *mbx* implements the original format, which is sometimes referred to as *mbxo*. This means that the *Content-Length* header, if present, is ignored and that any occurrences of « From » at the beginning of a line in a message body are transformed to « >From » when storing the message, although occurrences of « >From » are not transformed to « From » when reading the message.

Some *Mailbox* methods implemented by *mbx* deserve special remarks :

get_file (*key*)

Using the file after calling *flush()* or *close()* on the *mbx* instance may yield unpredictable results or raise an exception.

lock ()

unlock ()

Three locking mechanisms are used—dot locking and, if available, the *flock()* and *lockf()* system calls.

Voir aussi :

mbx man page from gmail A specification of the format and its variations.

mbx man page from tin Another specification of the format, with details on locking.

Configuring Netscape Mail on Unix : Why The Content-Length Format is Bad An argument for using the original mbox format rather than a variation.

« mbox » is a family of several mutually incompatible mailbox formats A history of mbox variations.

MH

class mailbox.**MH** (*path*, *factory*=None, *create*=True)

A subclass of *Mailbox* for mailboxes in MH format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is None, *MHMessage* is used as the default message representation. If *create* is True, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called *folders*) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support *sequences*, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called *.mh_sequences* in each folder.

The *MH* class manipulates MH mailboxes, but it does not attempt to emulate all of *mh*'s behaviors. In particular, it does not modify and is not affected by the *context* or *.mh_profile* files that are used by *mh* to store its state and configuration.

MH instances have all of the methods of *Mailbox* in addition to the following :

list_folders ()

Return a list of the names of all folders.

get_folder (*folder*)

Return an *MH* instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder (*folder*)

Create a folder whose name is *folder* and return an *MH* instance representing it.

remove_folder (*folder*)

Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

get_sequences ()

Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

set_sequences (*sequences*)

Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by *get_sequences* ().

pack ()

Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

Note : Already-issued keys are invalidated by this operation and should not be subsequently used.

Some *Mailbox* methods implemented by *MH* deserve special remarks :

remove (*key*)

__delitem__ (*key*)

discard (*key*)

These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

lock ()

unlock ()

Three locking mechanisms are used—dot locking and, if available, the *flock* () and *lockf* () system calls. For MH mailboxes, locking the mailbox means locking the *.mh_sequences* file and, only for the duration of any operations that affect them, locking individual message files.

get_file (*key*)

Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

flush()

All changes to MH mailboxes are immediately applied, so this method does nothing.

close()

MH instances do not keep any open files, so this method is equivalent to *unlock()*.

Voir aussi :

nmh - Message Handling System Home page of **nmh**, an updated version of the original **mh**.

MH & nmh : Email for Users & Programmers A GPL-licensed book on **mh** and **nmh**, with some information on the mailbox format.

Babyl

class mailbox.Babyl (*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in Babyl format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is *None*, *BabylMessage* is used as the default message representation. If *create* is *True*, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore ('`\037`') and Control-L ('`\014`'). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore ('`\037`') character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

Babyl instances have all of the methods of *Mailbox* in addition to the following :

get_labels()

Return a list of the names of all user-defined labels used in the mailbox.

Note : The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some *Mailbox* methods implemented by *Babyl* deserve special remarks :

get_file (*key*)

In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into an *io.BytesIO* instance, which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

lock()

unlock()

Three locking mechanisms are used—dot locking and, if available, the *flock()* and *lockf()* system calls.

Voir aussi :

Format of Version 5 Babyl Files A specification of the Babyl format.

Reading Mail with Rmail The Rmail manual, with some information on Babyl semantics.

MMDF

class mailbox.**MMDF** (*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in MMDF format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is *None*, *MMDFMessage* is used as the default message representation. If *create* is *True*, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A ('`\001`') characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are « From », but additional occurrences of « From » are not transformed to « >From » when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some *Mailbox* methods implemented by *MMDF* deserve special remarks :

get_file (*key*)

Using the file after calling *flush()* or *close()* on the *MMDF* instance may yield unpredictable results or raise an exception.

lock ()

unlock ()

Three locking mechanisms are used—dot locking and, if available, the *flock()* and *lockf()* system calls.

Voir aussi :

mmdf man page from tin A specification of MMDF format from the documentation of tin, a newsreader.

MMDF A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

19.4.2 Message objects

class mailbox.**Message** (*message=None*)

A subclass of the *email.message* module's *Message*. Subclasses of *mailbox.Message* add mailbox-format-specific state and behavior.

If *message* is omitted, the new instance is created in a default, empty state. If *message* is an *email.message.Message* instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if *message* is a *Message* instance. If *message* is a string, a byte string, or a file, it should contain an **RFC 2822**-compliant message, which is read and parsed. Files should be open in binary mode, but text mode files are accepted for backward compatibility.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that *Message* instances be used to represent messages retrieved using *Mailbox* instances. In some situations, the time and memory required to generate *Message* representations might not be acceptable. For such situations, *Mailbox* instances also offer string and file-like representations, and a custom message factory may be specified when a *Mailbox* instance is initialized.

MaildirMessage**class** mailbox.**MaildirMessage** (*message=None*)

A message with Maildir-specific behaviors. Parameter *message* has the same meaning as with the [Message](#) constructor.

Typically, a mail user agent application moves all of the messages in the `new` subdirectory to the `cur` subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in `cur` has an « info » section added to its file name to store information about its state. (Some mail readers may also add an « info » section to messages in `new`.) The « info » section may take one of two forms : it may contain « 2, » followed by a list of standardized flags (e.g., « 2,FR ») or it may contain « 1, » followed by so-called experimental information. Standard flags for Maildir messages are as follows :

Option	Signification	Explication
D	Draft	Under composition
F	Flagged	Marked as important
P	Passed	Forwarded, resent, or bounced
R	Replied	Replied to
S	Seen	Read
T	Trashed	Marked for subsequent deletion

[MaildirMessage](#) instances offer the following methods :

get_subdir ()

Return either « new » (if the message should be stored in the `new` subdirectory) or « cur » (if the message should be stored in the `cur` subdirectory).

Note : A message is typically moved from `new` to `cur` after its mailbox has been accessed, whether or not the message is has been read. A message `msg` has been read if "S" in `msg.get_flags()` is True.

set_subdir (*subdir*)

Set the subdirectory the message should be stored in. Parameter *subdir* must be either « new » or « cur ».

get_flags ()

Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of 'D', 'F', 'P', 'R', 'S', and 'T'. The empty string is returned if no flags are set or if « info » contains experimental semantics.

set_flags (*flags*)

Set the flags specified by *flags* and unset all others.

add_flag (*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character. The current « info » is overwritten whether or not it contains experimental information rather than flags.

remove_flag (*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character. If « info » contains experimental information rather than flags, the current « info » is not modified.

get_date ()

Return the delivery date of the message as a floating-point number representing seconds since the epoch.

set_date (*date*)

Set the delivery date of the message to *date*, a floating-point number representing seconds since the epoch.

get_info ()

Return a string containing the « info » for a message. This is useful for accessing and modifying « info » that is experimental (i.e., not a list of flags).

set_info (*info*)

Set « info » to *info*, which should be a string.

When a *MaildirMessage* instance is created based upon an *mbxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place :

Resulting state	<i>mbxMessage</i> or <i>MMDFMessage</i> state
« cur » subdirectory	O flag
F flag	F flag
R flag	A flag
S flag	R flag
T flag	D flag

When a *MaildirMessage* instance is created based upon an *MHMessage* instance, the following conversions take place :

Resulting state	<i>MHMessage</i> state
« cur » subdirectory	« unseen » sequence
« cur » subdirectory and S flag	no « unseen » sequence
F flag	« flagged » sequence
R flag	« replied » sequence

When a *MaildirMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place :

Resulting state	<i>BabylMessage</i> state
« cur » subdirectory	« unseen » label
« cur » subdirectory and S flag	no « unseen » label
P flag	« forwarded » or « resent » label
R flag	« answered » label
T flag	« deleted » label

mbxMessage**class** mailbox.**mbxMessage** (*message=None*)

A message with mbox-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Messages in an mbox mailbox are stored together in a single file. The sender's envelope address and the time of delivery are typically stored in a line beginning with « From » that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in *Status* and *X-Status* headers.

Conventional flags for mbox messages are as follows :

Option	Signification	Explication
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The « R » and « O » flags are stored in the *Status* header, and the « D », « F », and « A » flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

mbboxMessage instances offer the following methods :

get_from()

Return a string representing the « From » line that marks the start of the message in an mbox mailbox. The leading « From » and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

Set the « From » line to *from_*, which should be specified without a leading « From » or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *time.struct_time* instance, a tuple suitable for passing to *time.strftime()*, or True (to use *time.gmtime()*).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of 'R', 'O', 'D', 'F', and 'A'.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of 'R', 'O', 'D', 'F', and 'A'.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an *mbboxMessage* instance is created based upon a *MaiDirMessage* instance, a « From » line is generated based upon the *MaiDirMessage* instance's delivery date, and the following conversions take place :

Resulting state	<i>MaiDirMessage</i> state
R flag	S flag
O flag	« cur » subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an *mbboxMessage* instance is created based upon an *MHMessage* instance, the following conversions take place :

Resulting state	<i>MHMessage</i> state
R flag and O flag	no « unseen » sequence
O flag	« unseen » sequence
F flag	« flagged » sequence
A flag	« replied » sequence

When an *mbboxMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place :

Resulting state	<i>BabylMessage</i> state
R flag and O flag	no « unseen » label
O flag	« unseen » label
D flag	« deleted » label
A flag	« answered » label

When a *Message* instance is created based upon an *MMDFMessage* instance, the « From » line is copied and all flags directly correspond :

Resulting state	<i>MMDFMessage</i> state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

MHMessage

class mailbox.*MHMessage* (*message=None*)

A message with MH-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard *mh* and *nmh*) use sequences in much the same way flags are used with other formats, as follows :

Sequence	Explication
unseen	Not read, but previously detected by MUA
replied	Replied to
flagged	Marked as important

MHMessage instances offer the following methods :

get_sequences ()

Return a list of the names of sequences that include this message.

set_sequences (*sequences*)

Set the list of sequences that include this message.

add_sequence (*sequence*)

Add *sequence* to the list of sequences that include this message.

remove_sequence (*sequence*)

Remove *sequence* from the list of sequences that include this message.

When an *MHMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place :

Resulting state	<i>MaildirMessage</i> state
« unseen » sequence	no S flag
« replied » sequence	R flag
« flagged » sequence	F flag

When an *MHMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place :

Resulting state	<i>mboxMessage</i> or <i>MMDFMessage</i> state
« unseen » sequence	no R flag
« replied » sequence	A flag
« flagged » sequence	F flag

When an *MHMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place :

Resulting state	<i>BabylMessage</i> state
« unseen » sequence	« unseen » label
« replied » sequence	« answered » label

BabylMessage

class mailbox.**BabylMessage** (*message=None*)

A message with Babyl-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows :

Label	Explication
unseen	Not read, but previously detected by MUA
deleted	Marked for subsequent deletion
filed	Copied to another file or mailbox
answered	Replied to
forwarded	Forwarded
edited	Modified by the user
resent	Resent

By default, Rmail displays only visible headers. The *BabylMessage* class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

BabylMessage instances offer the following methods :

get_labels ()

Return a list of labels on the message.

set_labels (*labels*)

Set the list of labels on the message to *labels*.

add_label (*label*)

Add *label* to the list of labels on the message.

remove_label (*label*)

Remove *label* from the list of labels on the message.

get_visible ()

Return an *Message* instance whose headers are the message's visible headers and whose body is empty.

set_visible (*visible*)

Set the message's visible headers to be the same as the headers in *message*. Parameter *visible* should be a *Message* instance, an *email.message.Message* instance, a string, or a file-like object (which should be open in text mode).

update_visible ()

When a *BabylMessage* instance's original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows : each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of *Date*, *From*, *Reply-To*, *To*, *CC*, and *Subject* that are present in the original headers but not the visible headers are added to the visible headers.

When a *BabylMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place :

Resulting state	<i>MaiDirMessage</i> state
« unseen » label	no S flag
« deleted » label	T flag
« answered » label	R flag
« forwarded » label	P flag

When a *BabylMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place :

Resulting state	<i>mboxMessage</i> or <i>MMDFMessage</i> state
« unseen » label	no R flag
« deleted » label	D flag
« answered » label	A flag

When a *BabylMessage* instance is created based upon an *MHMessage* instance, the following conversions take place :

Resulting state	<i>MHMessage</i> state
« unseen » label	« unseen » sequence
« answered » label	« replied » sequence

MMDFMessage

class mailbox.**MMDFMessage** (*message=None*)

A message with MMDF-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender's address and the delivery date in an initial line beginning with « From ». Likewise, flags that indicate the state of the message are typically stored in *Status* and *X-Status* headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows :

Option	Signification	Explication
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The « R » and « O » flags are stored in the *Status* header, and the « D », « F », and « A » flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

MMDFMessage instances offer the following methods, which are identical to those offered by *mboxMessage* :

get_from ()

Return a string representing the « From » line that marks the start of the message in an mbox mailbox. The leading « From » and the trailing newline are excluded.

set_from (*from_*, *time_=None*)

Set the « From » line to *from_*, which should be specified without a leading « From » or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *time.struct_time* instance, a tuple suitable for passing to *time.strftime* (), or True (to use *time.gmtime* ()).

get_flags ()

Return a string specifying the flags that are currently set. If the message complies with the conventional format,

the result is the concatenation in the following order of zero or one occurrence of each of 'R', 'O', 'D', 'F', and 'A'.

set_flags (*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of 'R', 'O', 'D', 'F', and 'A'.

add_flag (*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag (*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an *MMDFMessage* instance is created based upon a *MaildirMessage* instance, a « From » line is generated based upon the *MaildirMessage* instance's delivery date, and the following conversions take place :

Resulting state	<i>MaildirMessage</i> state
R flag	S flag
O flag	« cur » subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an *MMDFMessage* instance is created based upon an *MHMessage* instance, the following conversions take place :

Resulting state	<i>MHMessage</i> state
R flag and O flag	no « unseen » sequence
O flag	« unseen » sequence
F flag	« flagged » sequence
A flag	« replied » sequence

When an *MMDFMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place :

Resulting state	<i>BabylMessage</i> state
R flag and O flag	no « unseen » label
O flag	« unseen » label
D flag	« deleted » label
A flag	« answered » label

When an *MMDFMessage* instance is created based upon an *mbxMessage* instance, the « From » line is copied and all flags directly correspond :

Resulting state	<i>mbxMessage</i> state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

19.4.3 Exceptions

The following exception classes are defined in the `mailbox` module :

exception `mailbox.Error`

The based class for all other module-specific exceptions.

exception `mailbox.NoSuchMailboxError`

Raised when a mailbox is expected but is not found, such as when instantiating a `Mailbox` subclass with a path that does not exist (and with the `create` parameter set to `False`), or when opening a folder that does not exist.

exception `mailbox.NotEmptyError`

Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

exception `mailbox.ExternalClashError`

Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely-generated file name already exists.

exception `mailbox.FormatError`

Raised when the data in a file cannot be parsed, such as when an `MH` instance attempts to read a corrupted `.mh_sequences` file.

19.4.4 Examples

A simple example of printing the subjects of all messages in a mailbox that seem interesting :

```
import mailbox
for message in mailbox.mbox('~/.mbox'):
    subject = message['subject']          # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted :

```
import mailbox
destination = mailbox.MH('~/.Mail')
destination.lock()
for message in mailbox.Babyl('~/.RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox :

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/.email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/.Maildir', factory=None)

for key in inbox.iterkeys():
```

(suite sur la page suivante)

(suite de la page précédente)

```
try:
    message = inbox[key]
except email.errors.MessageParseError:
    continue          # The message is malformed. Just leave it.

for name in list_names:
    list_id = message['list-id']
    if list_id and name in list_id:
        # Get mailbox to use
        box = boxes[name]

        # Write copy to disk before removing original.
        # If there's a crash, you might duplicate a message, but
        # that's better than losing a message completely.
        box.lock()
        box.add(message)
        box.flush()
        box.unlock()

        # Remove original message
        inbox.lock()
        inbox.discard(key)
        inbox.flush()
        inbox.unlock()
        break          # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

19.5 mimetypes — Map filenames to MIME types

Source code : [Lib/mimetypes.py](#)

The `mimetypes` module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call `init()` if they rely on the information `init()` sets up.

`mimetypes.guess_type(url, strict=True)`

Guess the type of a file based on its filename or URL, given by `url`. The return value is a tuple (`type`, `encoding`) where `type` is `None` if the type can't be guessed (missing or unknown suffix) or a string of the form 'type/subtype', usable for a MIME `content-type` header.

`encoding` is `None` for no encoding or the name of the program used to encode (e.g. **compress** or **gzip**). The encoding is suitable for use as a `Content-Encoding` header, **not** as a `Content-Transfer-Encoding` header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case sensitively, then case insensitively.

The optional *strict* argument is a flag specifying whether the list of known MIME types is limited to only the official types registered with IANA. When *strict* is `True` (the default), only the IANA types are supported; when *strict* is `False`, some additional non-standard but commonly used MIME types are also recognized.

`mimetypes.guess_all_extensions (type, strict=True)`

Guess the extensions for a file based on its MIME type, given by *type*. The return value is a list of strings giving all possible filename extensions, including the leading dot ('.'). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by `guess_type()`.

The optional *strict* argument has the same meaning as with the `guess_type()` function.

`mimetypes.guess_extension (type, strict=True)`

Guess the extension for a file based on its MIME type, given by *type*. The return value is a string giving a filename extension, including the leading dot ('.'). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by `guess_type()`. If no extension can be guessed for *type*, `None` is returned.

The optional *strict* argument has the same meaning as with the `guess_type()` function.

Some additional functions and data items are available for controlling the behavior of the module.

`mimetypes.init (files=None)`

Initialize the internal data structures. If given, *files* must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from `knownfiles`; on Windows, the current registry settings are loaded. Each file named in *files* or `knownfiles` takes precedence over those named before it. Calling `init()` repeatedly is allowed.

Specifying an empty list for *files* will prevent the system defaults from being applied: only the well-known values will be present from a built-in list.

Modifié dans la version 3.2 : Previously, Windows registry settings were ignored.

`mimetypes.read_mime_types (filename)`

Load the type map given in the file *filename*, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot ('. '), to strings of the form 'type/subtype'. If the file *filename* does not exist or cannot be read, `None` is returned.

`mimetypes.add_type (type, ext, strict=True)`

Add a mapping from the MIME type *type* to the extension *ext*. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is `True` (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

`mimetypes.inited`

Flag indicating whether or not the global data structures have been initialized. This is set to `True` by `init()`.

`mimetypes.knownfiles`

List of type map file names commonly installed. These files are typically named `mime.types` and are installed in different locations by different packages.

`mimetypes.suffix_map`

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately.

`mimetypes.encodings_map`

Dictionary mapping filename extensions to encoding types.

`mimetypes.types_map`

Dictionary mapping filename extensions to MIME types.

`mimetypes.common_types`

Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

Un exemple d'utilisation du module :

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

19.5.1 MimeTypes Objects

The *MimeTypes* class may be useful for applications which may want more than one MIME-type database ; it provides an interface similar to the one of the *mimetypes* module.

class *mimetypes.MimeTypes* (*filenames=()*, *strict=True*)

This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional *mime.types*-style files into the database using the *read()* or *readfp()* methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional *filenames* parameter can be used to cause additional files to be loaded « on top » of the default database.

suffix_map

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the *.tgz* extension is mapped to *.tar.gz* to allow the encoding and type to be recognized separately. This is initially a copy of the global *suffix_map* defined in the module.

encodings_map

Dictionary mapping filename extensions to encoding types. This is initially a copy of the global *encodings_map* defined in the module.

types_map

Tuple containing two dictionaries, mapping filename extensions to MIME types : the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by *common_types* and *types_map*.

types_map_inv

Tuple containing two dictionaries, mapping MIME types to a list of filename extensions : the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by *common_types* and *types_map*.

guess_extension (*type*, *strict=True*)

Similar to the *guess_extension()* function, using the tables stored as part of the object.

guess_type (*url*, *strict=True*)

Similar to the *guess_type()* function, using the tables stored as part of the object.

guess_all_extensions (*type*, *strict=True*)

Similar to the *guess_all_extensions()* function, using the tables stored as part of the object.

read (*filename*, *strict=True*)

Load MIME information from a file named *filename*. This uses *readfp()* to parse the file.

If *strict* is *True*, information will be added to list of standard types, else to the list of non-standard types.

readfp (*fp*, *strict=True*)

Load MIME type information from an open file *fp*. The file must have the format of the standard *mime.types* files.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

read_windows_registry (*strict*=`True`)

Load MIME type information from the Windows registry. Availability : Windows.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

Nouveau dans la version 3.2.

19.6 base64 — Encodages base16, base32, base64 et base85

Code source : [Lib/base64.py](#)

Ce module fournit des fonctions permettant de coder des données binaires en caractères ASCII affichables ainsi que de décoder ces caractères vers des données binaires en retour. Il fournit des fonctions d'encodage et de décodage pour les codages spécifiés par la [RFC 3548](#) qui définit les algorithmes base16, base32 et base64, ainsi que les encodages standards *de facto* Ascii85 et base85.

Les encodages définis par la [RFC 3548](#) sont adaptés au codage des données binaires pour leur transfert par courriel, comme éléments d'une URL ou d'une requête HTTP POST. L'algorithme d'encodage ne doit pas être confondu avec le programme **uuencode**.

Ce module présente deux interfaces. L'interface moderne gère l'encodage d'*objets octet-compatibles* en *bytes* ASCII ainsi que le décodage d'*objets octet-compatibles* ou de chaînes de caractères contenant de l'ASCII en *bytes*. Les deux alphabets de l'algorithme base64 définis par la [RFC 3548](#) (normal et sûr pour les systèmes de fichiers ou URL) sont gérés.

L'interface historique ne permet pas le décodage des chaînes de caractères mais fournit des fonctions permettant d'encoder et décoder depuis et vers des *objets fichiers*. Elle ne gère que l'alphabet base64 standard et ajoute une nouvelle ligne tous les 76 caractères, comme spécifié par la [RFC 2045](#). Notez que le paquet *email* est probablement ce que vous cherchez si vous souhaitez une implémentation de la [RFC 2045](#).

Modifié dans la version 3.3 : Les chaînes de caractères Unicode contenant uniquement des caractères ASCII sont désormais acceptées par les fonctions de décodage de l'interface moderne.

Modifié dans la version 3.4 : Tous les *objets octet-compatibles* sont désormais acceptés par l'ensemble des fonctions d'encodage et de décodage de ce module. La gestion de Ascii85/base85 a été ajoutée.

L'interface moderne propose :

`base64.b64encode` (*s*, *altchars*=`None`)

Encode un *objet octet-compatible* *s* en utilisant l'algorithme base64 et renvoie les *bytes* encodés.

L'option *altchars* doit être un *bytes-like object* de longueur au moins 2 (les caractères additionnels sont ignorés) qui spécifie un alphabet alternatif pour les délimiteurs + et /. Cela permet de générer des chaînes de caractères base64 pouvant être utilisées pour une URL ou dans un système de fichiers. La valeur par défaut est `None`, auquel cas l'alphabet standard base64 est utilisé.

`base64.b64decode` (*s*, *altchars*=`None`, *validate*=`False`)

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* encodée en base64 et renvoie les *bytes* décodés.

L'option *altchars* doit être un *bytes-like object* de longueur au moins égale à 2 (les caractères additionnels sont ignorés) qui spécifie un alphabet alternatif pour les délimiteurs + et /.

Une exception `binascii.Error` est levée si *s* n'est pas remplie à une longueur attendue.

Si *validate* est `False` (par défaut), les caractères qui ne sont ni dans l'alphabet base64 normal, ni dans l'alphabet alternatif, sont ignorés avant la vérification de la longueur du remplissage. Si *validate* est `True`, les caractères hors de l'alphabet de l'entrée produisent une `binascii.Error`.

`base64.standard_b64encode` (*s*)

Encode un *objet octet-compatible* *s* en utilisant l'alphabet standard base64 et renvoie les *bytes* encodés.

`base64.standard_b64decode(s)`

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* utilisant l'alphabet base64 standard et renvoie les *bytes* décodés.

`base64.urlsafe_b64encode(s)`

Encode un *objet byte-compatible* *s* en utilisant un alphabet sûr pour les URL et systèmes de fichiers qui substitue `-` et `_` à `+` et `/` dans l'alphabet standard base64 et renvoie les *bytes* encodés.

`base64.urlsafe_b64decode(s)`

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* utilisant un alphabet sûr pour les URL et systèmes de fichiers qui substitue `-` et `_` à `+` et `/` dans l'alphabet standard base64 et renvoie les *bytes* décodés.

`base64.b32encode(s)`

Encode un *objet byte-compatible* *s* en utilisant l'algorithme base32 et renvoie les *bytes* encodés.

`base64.b32decode(s, casefold=False, map01=None)`

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* encodé en base32 et renvoie les *bytes* décodés.

L'option *casefold* est un drapeau spécifiant si l'utilisation d'un alphabet en minuscules est acceptable comme entrée. Pour des raisons de sécurité, cette option est à `False` par défaut.

La [RFC 3548](#) autorise une correspondance optionnelle du chiffre 0 (zéro) vers la lettre O (/o/) ainsi que du chiffre 1 (un) vers soit la lettre I (/i/) ou la lettre L (/l/). L'argument optionnel *map01*, lorsqu'il diffère de `None`, spécifie en quelle lettre le chiffre 1 doit être transformé (lorsque *map01* n'est pas `None`, le chiffre 0 est toujours transformé en la lettre O). Pour des raisons de sécurité, le défaut est `None`, de telle sorte que 0 et 1 ne sont pas autorisés dans l'entrée.

Une exception `binascii.Error` est levée si *s* n'est pas remplie à une longueur attendue ou si elle contient des caractères hors de l'alphabet.

`base64.b16encode(s)`

Encode un *objet byte-compatible* *s* en utilisant l'algorithme base16 et renvoie les *bytes* encodés.

`base64.b16decode(s, casefold=False)`

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* encodé en base16 et renvoie les *bytes* décodés.

L'option *casefold* est un drapeau spécifiant si l'utilisation d'un alphabet en minuscules est acceptable comme entrée. Pour des raisons de sécurité, cette option est à `False` par défaut.

Une exception `binascii.Error` est levée si *s* n'est pas remplie à une longueur attendue ou si elle contient des caractères hors de l'alphabet.

`base64.a85encode(b, *, foldspaces=False, wrapcol=0, pad=False, adobe=False)`

Encode un *objet byte-compatible* *s* en utilisant l'algorithme Ascii85 et renvoie les *bytes* encodés.

L'option *foldspaces* permet d'utiliser la séquence spéciale "y" à la place de quatre espaces consécutives (ASCII 0x20) comme pris en charge par *btoa*. Cette fonctionnalité n'est pas gérée par l'encodage « standard » Ascii85.

wrapcol contrôle l'ajout de caractères de saut de ligne (`b'\n'`) à la sortie. Chaque ligne de sortie contient au maximum *wrapcol* caractères si cette option diffère de zéro.

pad spécifie l'ajout de caractères de remplissage (*padding* en anglais) à l'entrée jusqu'à ce que sa longueur soit un multiple de 4 avant encodage. Notez que l'implémentation *btoa* effectue systématiquement ce remplissage.

adobe contrôle si oui ou non la séquence encodée d'octets est encadrée par `<~` et `~>` comme utilisé dans l'implémentation Adobe.

Nouveau dans la version 3.4.

`base64.a85decode(b, *, foldspaces=False, adobe=False, ignorechars=b'\t\n\r\v')`

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* encodé en Ascii85 et renvoie les *bytes* décodés.

L'option *foldspaces* permet d'utiliser la séquence spéciale "y" à la place de quatre espaces consécutives (ASCII 0x20) comme pris en charge par *btoa*. Cette fonctionnalité n'est pas gérée par l'encodage « standard » Ascii85.

adobe indique si la séquence d'entrée utilise le format Adobe Ascii85 (c'est-à-dire utilise l'encadrement par <~ et ~>).

ignorechars doit être un *bytes-like object* ou une chaîne ASCII contenant des caractères à ignorer dans l'entrée. Il ne doit contenir que des caractères d'espacement et contient par défaut l'ensemble des caractères d'espacement de l'alphabet ASCII.

Nouveau dans la version 3.4.

`base64.b85encode(b, pad=False)`

Encode un *objet byte-compatible* *s* en utilisant l'algorithme base85 (tel qu'utilisé par exemple par le programme *git-diff* sur des données binaires) et renvoie les *bytes* encodés.

Si *pad* est vrai, des caractères de remplissage `b'``0'` (*padding* en anglais) sont ajoutés à l'entrée jusqu'à ce que sa longueur soit un multiple de 4 octets avant encodage.

Nouveau dans la version 3.4.

`base64.b85decode(b)`

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *b* encodé en base85 et renvoie les *bytes* décodés. Les caractères de remplissage sont implicitement retirés si nécessaire.

Nouveau dans la version 3.4.

L'interface historique :

`base64.decode(input, output)`

Décode le contenu d'un fichier binaire *input* et écrit les données binaires résultantes dans le fichier *output*. *input* et *output* doivent être des *objets fichiers*. *input* est lu jusqu'à ce que `input.readline()` renvoie un objet *bytes* vide.

`base64.decodebytes(s)`

Décode un *objet octet-compatible* *s* devant contenir une ou plusieurs lignes de données encodées en base64 et renvoie les *bytes* décodés.

Nouveau dans la version 3.1.

`base64.decodestring(s)`

Alias obsolète de `decodebytes()`.

Obsolète depuis la version 3.1.

`base64.encode(input, output)`

Encode le contenu du fichier binaire *input* et écrit les données encodées en base64 résultantes dans le fichier *output*. **input* et *output* doivent être des *objets fichiers*. *input* est lu jusqu'à ce que `input.readline()` renvoie un objet *bytes* vide. `encode()` insère un caractère de saut de ligne (`b'\n'`) tous les 76 octets de sortie et assure que celle-ci se termine par une nouvelle ligne, comme spécifié par la [RFC 2045](#) (MIME).

`base64.encodebytes(s)`

Encode un *objet octet-compatible* *s* pouvant contenir des données binaires arbitraires et renvoie les *bytes* contenant les données encodées en base64. Un caractère de saut de ligne (`b'\n'`) est inséré tous les 76 octets de sortie et celle-ci se termine par une nouvelle ligne, comme spécifié par la [RFC 2045](#) (MIME).

Nouveau dans la version 3.1.

`base64.encodestring(s)`

Alias obsolète de `encodebytes()`.

Obsolète depuis la version 3.1.

Un exemple d'utilisation du module :

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSB1bmNvZGVk'
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

Voir aussi :

Module *binascii* Module secondaire contenant les conversions ASCII vers binaire et binaire vers ASCII.

RFC 1521 — MIME (Multipurpose Internet Mail Extensions) Part One : Mechanisms for Specifying and Describing the Format of

La Section 5.2, « *Base64 Content-Transfer-Encoding* », donne la définition de l'encodage base64.

19.7 binhex — Encode et décode les fichiers *binhex4*

Code source : [Lib/binhex.py](#)

Ce module encode et décode les fichiers au format *binhex4*, un format permettant la représentation de fichier Macintosh au format ASCII. Seulement la fourchette de donnée est supportée.

Le module *binhex* définit les fonctions suivantes :

`binhex.binhex(input, output)`

Convertit un fichier binaire avec comme nom *input* en fichier *binhex output*. Le paramètre *output* peut être soit un nom de fichier, soit un objet s'apparentant à un fichier (tout objet supportant les méthodes `write()` et `close()`).

`binhex.hexbin(input, output)`

Décode un fichier *binhex input*. *input* peut être soit un nom de fichier, soit un objet s'apparentant à un fichier supportant les méthodes `write()` et `close()`. Le résultat est écrit dans un fichier nommé *output*, sauf si l'argument est `None`, dans ce cas le fichier de sortie est lu depuis le fichier *binhex*.

L'exception suivante est aussi définie :

exception `binhex.Error`

Exception levée quand quelque chose ne peut être encodé en utilisant le format *binhex* (par exemple, un nom de fichier trop long pour rentrer dans le champ *filename*) ou quand les données d'entrée ne sont pas encodées correctement en *binhex*.

Voir aussi :

Module *binascii* Module secondaire contenant les conversions ASCII vers binaire et binaire vers ASCII.

19.7.1 Notes

Il y a une alternative, une interface plus puissante pour le codeur et décodeur, voir les sources pour les détails.

Si vous codez ou décidez sur des plateformes autres que Macintosh, elles utiliseront l'ancienne convention Macintosh pour les retours à la ligne (retour-chariot comme fin de ligne).

19.8 binascii — Conversion entre binaire et ASCII

Le module `binascii` contient des méthodes pour convertir entre binaire et diverses représentations binaires encodées en ASCII. Normalement, vous n'allez pas utiliser ces fonctions directement mais vous utiliserez des modules d'encapsulation comme `uu`, `base64`, or `binhex` à la place. Le module `binascii` contient des fonctions bas-niveau écrites en C plus rapides qui sont utilisées par des modules haut-niveau.

Note : La fonction `a2b_*` accepte des chaînes de caractères contenant seulement des caractères ASCII. D'autres fonctions acceptent seulement des objets *bytes et similaire* (tel que `bytes`, `bytearray` et autres objets qui supportent le protocole tampon).

Modifié dans la version 3.3 : Les chaînes de caractères *unicode* seulement composées de caractères ASCII sont désormais acceptées par les fonctions `a2b_*`.

Le module `binascii` définit les fonctions suivantes :

`binascii.a2b_uu(string)`

Convertit une seule ligne de donnée *uuencoded* en binaire et renvoie la donnée binaire. Les lignes contiennent normalement 45 octets (binaire), sauf pour la dernière ligne. Il se peut que la ligne de donnée soit suivie d'un espace blanc.

`binascii.b2a_uu(data)`

Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of *data* should be at most 45.

`binascii.a2b_base64(string)`

Convertit un bloc de donnée en *base64* en binaire et renvoie la donnée binaire. Plus d'une ligne peut être passé à la fois.

`binascii.b2a_base64(data, *, newline=True)`

Convertit les données binaires en une ligne de caractères ASCII en codage base 64. La valeur de renvoyée et la ligne convertie, incluant un caractère de nouvelle ligne si *newline* est vraie. La sortie de cette fonction se conforme à [RFC 3548](#).

Modifié dans la version 3.6 : Ajout du paramètre *newline*.

`binascii.a2b_qp(data, header=False)`

Convertit un bloc de données *quoted-printable* en binaire et renvoie les données binaires. Plus d'une ligne peut être passée à la fois. Si l'argument optionnel *header* est présent et vrai, les traits soulignés seront décodés en espaces.

`binascii.b2a_qp(data, quotetabs=False, istext=True, header=False)`

Convertit les données binaires en ligne(s) de caractères ASCII en codage imprimable entre guillemets. La valeur de retour est la ligne(s) convertie(s). Si l'argument optionnel *quotetabs* est présent et vrai, toutes les tabulations et espaces seront encodés. Si l'argument optionnel *istext* est présent et faux, les nouvelles lignes ne sont pas encodées mais les espaces de fin de ligne le seront. Si l'argument optionnel *header* est présent et vrai, les espaces vont être encodés comme de traits soulignés selon [RFC 1522](#). Si l'argument optionnel *header* est présent et faux, les caractères de nouvelle ligne seront également encodés ; sinon la conversion de saut de ligne pourrait corrompre le flux de données binaire.

`binascii.a2b_hqx(string)`

Convertit un bloc de donnée ASCII au format *binhex4* en binaire, sans faire de décompression RLE. La chaîne de caractères doit contenir un nombre complet d'octet binaires ou (au cas où la dernière portion de donnée est au format *binhex4*) avoir les bits restants à 0.

`binascii.rledecode_hqx(data)`

Réalise une décompression RLE sur la donnée, d'après la norme *binhex4*. L'algorithme utilise 0x90 après un octet

comme un indicateur de répétition, suivi d'un décompte. Un décompte de 0 définit une valeur d'octet de 0x90. La routine renvoie la donnée décompressée, sauf si la donnée entrante se finit sur un indicateur de répétition orphelin. Dans ce cas l'exception *Incomplete* est levée.

Modifié dans la version 3.2 : Accepte seulement des objets *bytestring* ou *bytearray* en entrée.

`binascii.rlecode_hqx(data)`

Réalise une compression RLE de type *binhex4* sur *data* et renvoie le résultat.

`binascii.b2a_hqx(data)`

Réalise une traduction *hexbin4* de binaire à ASCII et renvoie la chaîne de caractères résultante. L'argument doit être *RLE-coded*, et avoir une longueur divisible par 3 (sauf, éventuellement, le dernier fragment).

`binascii.crc_hqx(data, value)`

Calcule une valeur en CRC 16-bit de *data*, commençant par *value* comme CRC initial et renvoie le résultat. Ceci utilise le CRC-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$, souvent représenté comme *0x1021*. Ce CRC est utilisé dans le format *binhex4*.

`binascii.crc32(data[, value])`

Calcule CRC-32, la somme de contrôle 32-bit de *data*, commençant par un CRC initial de *value*. Le CRC initial par défaut est zéro. L'algorithme est cohérent avec la somme de contrôle du fichier ZIP. Comme l'algorithme est conçu pour être utilisé comme un algorithme de somme de contrôle, il ne convient pas comme algorithme de hachage général. Utiliser comme suit :

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

Modifié dans la version 3.0 : Le résultat est toujours non signé. Pour générer la même valeur numérique sur toutes les versions de Python et plateformes, utilisez `crc32(data) & 0xffffffff`.

`binascii.b2a_hex(data)`

`binascii.hexlify(data)`

Renvoie la représentation hexadécimale du binaire *data*. Chaque octet de *data* est converti en la représentation 2 chiffres correspondante. L'objet octets renvoyé est donc deux fois plus long que la longueur de *data*.

`binascii.a2b_hex(hexstr)`

`binascii.unhexlify(hexstr)`

Renvoie la donnée binaire représentée par la chaîne de caractères hexadécimale *hexstr*. Cette fonction est l'inverse de *b2a_hex()*. *hexstr* doit contenir un nombre pair de chiffres hexadécimaux (qui peuvent être en majuscule ou minuscule), sinon une exception *Error* est levée.

exception `binascii.Error`

Exception levée en cas d'erreurs. Ce sont typiquement des erreurs de programmation.

exception `binascii.Incomplete`

Exception levée par des données incomplète. Il ne s'agit généralement pas d'erreurs de programmation, mais elles peuvent être traitées en lisant un peu plus de données et en réessayant.

Voir aussi :

Module *base64* Support de l'encodage *base64-style* conforme RFC en base 16, 32, 64 et 85.

Module *binhex* Support pour le format *binhex* utilisé sur Macintosh.

Module *uu* Gestion de l'encodage UU utilisé sur Unix.

Module *quopri* Support de l'encodage *quote-printable* utilisé par les messages *email* MIME.

19.9 quopri — Encode et décode des données *MIME quoted-printable*

Code source : [Lib/quopri.py](#)

Ce module effectue des encodages et décodages de transport *quoted-printable*, tel que définis dans la [RFC 1521](#) : « *MIME (Multipurpose Internet Mail Extensions) Part One Mechanisms for Specifying and Describing the Format of Internet Message Bodies* ». L'encodage *quoted-printable* est adapté aux données dans lesquelles peu de données ne sont pas affichables. L'encodage *base64* disponible dans le module [base64](#) est plus compact dans les cas où ces caractères sont nombreux, typiquement pour encoder des images.

`quopri.decode(input, output, header=False)`

Décode le contenu du fichier *input* et écrit le résultat décodé, binaire, dans le fichier *output*. *input* et *output* doivent être des *objets fichiers binaires*. Si l'argument facultatif *header* est fourni et vrai, les *underscores* seront décodés en espaces. C'est utilisé pour décoder des entêtes encodées « Q » décrits dans la RFC [RFC 1522](#) : « *MIME (Multipurpose Internet Mail Extensions) Part Two : Message Header Extensions for Non-ASCII Text* ».

`quopri.encode(input, output, quotetabs, header=False)`

Encode le contenu du fichier *input* et écrit le résultat dans le fichier *output*. *input* et *output* doivent être des *objets fichiers binaires*. *quotetabs* (paramètre obligatoire) permet de choisir le style d'encodage des espaces et des tabulations, si vrai les espaces seront encodées, sinon elles seront laissées inchangées. Notez que les espaces et tabulations en fin de ligne sont toujours encodées, tel que spécifié par la [RFC 1521](#). *header* est une option permettant d'encoder les espace en *underscores*, tel que spécifié par la [RFC 1522](#).

`quopri.decodestring(s, header=False)`

Fonctionne comme `decode()`, sauf qu'elle accepte des *bytes* comme source, et renvoie les *bytes* décodés correspondants.

`quopri.encodestring(s, quotetabs=False, header=False)`

Fonctionne comme `encode()`, sauf qu'elle accepte des *bytes* comme source et renvoie les *bytes* encodés correspondants. Par défaut, `False` est donné au paramètre *quotetabs* de la fonction `encode()`.

Voir aussi :

Module [base64](#) Encode et décode des données MIME en *base64*

19.10 uu — Encode et décode les fichiers *uuencode*

Code source : [Lib/uu.py](#)

Ce module encode et décode les fichiers au format *uuencode*, permettant à des données binaires d'être transférées lors de connexion ASCII. Pour tous les arguments où un fichier est attendu, les fonctions acceptent un « objet fichier-compatible ». Pour des raisons de compatibilité avec les anciennes versions de Python, une chaîne de caractères contenant un chemin est aussi acceptée, et le fichier correspondant sera ouvert en lecture et écriture ; le chemin `'-'` est considéré comme l'entrée ou la sortie standard. Cependant cette interface est obsolète ; il vaut mieux que l'appelant ouvre le fichier lui-même, en s'assurant si nécessaire que le mode d'ouverture soit `'rb'` ou `'wb'` sur Windows.

Ce code provient d'une contribution de Lance Ellinghouse et a été modifié par Jack Jansen.

Le module `uu` définit les fonctions suivantes :

`uu.encode(in_file, out_file, name=None, mode=None)`

Uuencode le fichier *in_file* dans le fichier *out_file*. Le fichier *uuencod  * contiendra une ent  te sp  cifiant les valeurs de *name* et *mode* par d  faut pour le d  codage du fichier. Par d  faut ces valeurs sont prises de *in_file* ou valent respectivement `'-'` et `0o666`.

`uu.decode(in_file, out_file=None, mode=None, quiet=False)`

D  code le fichier *in_file* et   crit le r  sultat dans *out_file*. Si *out_file* est un chemin, *mode* est utilis   pour les permissions du fichier lors de sa cr  ation. Les valeurs par d  faut pour *out_file* et *mode* sont r  cup  r  es des ent  tes *uuencode*. Cependant, si le fichier sp  cifi   par les ent  tes est d  j   existant, une exception `uu.Error` est lev  e.

La fonction *decode()*   crit un avertissement sur la sortie d'erreur si l'entr  e contient des erreurs mais que Python a pu s'en sortir. Mettre *quiet*    `True` emp   che l'  criture de cet avertissement.

exception `uu.Error`

Classe fille d'*Exception*, elle peut   tre lev  e par `uu.decode()` dans diff  rentes situations, tel que d  crit plus haut, mais aussi en cas d'ent  te mal format  e ou d'entr  e tronqu  e.

Voir aussi :

Module *binascii* Module secondaire contenant les conversions ASCII vers binaire et binaire vers ASCII.

Outils de traitement de balises structurées

Python intègre une variété de modules pour fonctionner avec différentes formes de données structurées et balisées, comme le SGML (*Standard Generalized Markup Language*), le HTML (*Hypertext Markup Language*), et quelques interfaces pour travailler avec du XML (*eXtensible Markup Language*).

20.1 `html` — Support du HyperText Markup Language

Source code : [Lib/html/__init__.py](#)

Ce module définit des outils permettant la manipulation d'HTML.

`html.escape(s, quote=True)`

Convertit les caractères `&`, `<` et `>` de la chaîne de caractères `s` en séquences HTML valides. À utiliser si le texte à afficher pourrait contenir de tels caractères dans le HTML. Si le paramètre optionnel `quote` est vrai, les caractères `"` et `'` sont également traduits ; cela est utile pour les inclusions dans des valeurs d'attributs HTML délimitées par des guillemets, comme dans ``.

Nouveau dans la version 3.2.

`html.unescape(s)`

Convert all named and numeric character references (e.g. `>`, `>`, `>`) in the string `s` to the corresponding Unicode characters. This function uses the rules defined by the HTML 5 standard for both valid and invalid character references, and the *list of HTML 5 named character references*.

Nouveau dans la version 3.4.

Les sous-modules dans le paquet `html` sont :

- `html.parser` – Parseur HTML/XHTML avec un mode de *parsing* tolérant
- `html.entities` – Définitions d'entités HTML

20.2 `html.parser` — Simple HTML and XHTML parser

Source code : <Lib/html/parser.py>

This module defines a class `HTMLParser` which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML.

class `html.parser.HTMLParser` (*, `convert_charrefs=True`)

Create a parser instance able to parse invalid markup.

If `convert_charrefs` is `True` (the default), all character references (except the ones in `script/style` elements) are automatically converted to the corresponding Unicode characters.

An `HTMLParser` instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass `HTMLParser` and override its methods to implement the desired behavior.

This parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

Modifié dans la version 3.4 : `convert_charrefs` keyword argument added.

Modifié dans la version 3.5 : The default value for argument `convert_charrefs` is now `True`.

20.2.1 Example HTML Parser Application

As a basic example, below is a simple HTML parser that uses the `HTMLParser` class to print out start tags, end tags, and data as they are encountered :

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
          '<body><h1>Parse me!</h1></body></html>')
```

The output will then be :

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

20.2.2 HTMLParser Methods

HTMLParser instances have the following methods :

`HTMLParser.feed(data)`

Feed some text to the parser. It is processed insofar as it consists of complete elements ; incomplete data is buffered until more data is fed or `close()` is called. *data* must be *str*.

`HTMLParser.close()`

Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the *HTMLParser* base class method `close()`.

`HTMLParser.reset()`

Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

`HTMLParser.getpos()`

Return current line number and offset.

`HTMLParser.get_starttag_text()`

Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML « as deployed » or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

The following methods are called when data or markup elements are encountered and they are meant to be overridden in a subclass. The base class implementations do nothing (except for `handle_startendtag()`) :

`HTMLParser.handle_starttag(tag, attrs)`

This method is called to handle the start of a tag (e.g. `<div id="main">`).

The *tag* argument is the name of the tag converted to lower case. The *attrs* argument is a list of (name, value) pairs containing the attributes found inside the tag's `<>` brackets. The *name* will be translated to lower case, and quotes in the *value* have been removed, and character and entity references have been replaced.

For instance, for the tag ``, this method would be called as `handle_starttag('a', [('href', 'https://www.cwi.nl/')])`.

All entity references from `html.entities` are replaced in the attribute values.

`HTMLParser.handle_endtag(tag)`

This method is called to handle the end tag of an element (e.g. `</div>`).

The *tag* argument is the name of the tag converted to lower case.

`HTMLParser.handle_startendtag(tag, attrs)`

Similar to `handle_starttag()`, but called when the parser encounters an XHTML-style empty tag (``). This method may be overridden by subclasses which require this particular lexical information; the default implementation simply calls `handle_starttag()` and `handle_endtag()`.

`HTMLParser.handle_data(data)`

This method is called to process arbitrary data (e.g. text nodes and the content of `<script>...</script>` and `<style>...</style>`).

`HTMLParser.handle_entityref(name)`

This method is called to process a named character reference of the form `&name;` (e.g. `>`), where *name* is a general entity reference (e.g. `'gt'`). This method is never called if `convert_charrefs` is `True`.

`HTMLParser.handle_charref(name)`

This method is called to process decimal and hexadecimal numeric character references of the form `&#NNN;` and `&#xNNN;`. For example, the decimal equivalent for `>` is `>`, whereas the hexadecimal is `>`; in this case the method will receive `'62'` or `'x3E'`. This method is never called if `convert_charrefs` is `True`.

`HTMLParser.handle_comment(data)`

This method is called when a comment is encountered (e.g. `<!--comment-->`).

For example, the comment `<!-- comment -->` will cause this method to be called with the argument `'comment '`.

The content of Internet Explorer conditional comments (condcoms) will also be sent to this method, so, for `<!--[if IE 9]>IE9-specific content<![endif]-->`, this method will receive `'[if IE 9]>IE9-specific content<![endif]'`.

`HTMLParser.handle_decl(decl)`

This method is called to handle an HTML doctype declaration (e.g. `<!DOCTYPE html>`).

The `decl` parameter will be the entire contents of the declaration inside the `<![...]>` markup (e.g. `'DOCTYPE html'`).

`HTMLParser.handle_pi(data)`

Method called when a processing instruction is encountered. The `data` parameter will contain the entire processing instruction. For example, for the processing instruction `<?proc color='red'>`, this method would be called as `handle_pi("proc color='red'")`. It is intended to be overridden by a derived class; the base class implementation does nothing.

Note : The `HTMLParser` class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing `'?'` will cause the `'?'` to be included in `data`.

`HTMLParser.unknown_decl(data)`

This method is called when an unrecognized declaration is read by the parser.

The `data` parameter will be the entire contents of the declaration inside the `<![...]>` markup. It is sometimes useful to be overridden by a derived class. The base class implementation does nothing.

20.2.3 Examples

The following class implements a parser that will be used to illustrate more examples :

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("    attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
        print("Data      :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
```

(suite sur la page suivante)

(suite de la page précédente)

```

        c = chr(int(name))
        print("Num ent  :", c)

    def handle_decl(self, data):
        print("Decl      :", data)

parser = MyHTMLParser()

```

Parsing a doctype :

```

>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl      : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
↳html4/strict.dtd"

```

Parsing an element with a few attributes and a title :

```

>>> parser.feed('')
Start tag: img
  attr: ('src', 'python-logo.png')
  attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data      : Python
End tag   : h1

```

The content of script and style elements is returned as is, without further parsing :

```

>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
  attr: ('type', 'text/css')
Data      : #python { color: green }
End tag   : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
  attr: ('type', 'text/javascript')
Data      : alert("<strong>hello!</strong>");
End tag   : script

```

Parsing comments :

```

>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment   : a comment
Comment   : [if IE 9]>IE-specific content<![endif]

```

Parsing named and numeric character references and converting them to the correct char (note : these 3 references are all equivalent to '>') :

```

>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent   : >
Num ent   : >

```

Feeding incomplete chunks to `feed()` works, but `handle_data()` might be called more than once (unless `convert_charrefs` is set to `True`):

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data      : buff
Data      : ered
Data      : text
End tag   : span
```

Parsing invalid HTML (e.g. unquoted attributes) also works :

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
      attr: ('class', 'link')
      attr: ('href', '#main')
Data      : tag soup
End tag   : p
End tag   : a
```

20.3 `html.entities` — Définitions des entités HTML générales

Source code : <Lib/html/entities.py>

Ce module définit quatre dictionnaires, `html5`, `name2codepoint`, `codepoint2name`, et `entitydefs`.

`html.entities.html5`

Un dictionnaire qui fait correspondre les références de caractères nommés HTML5¹ aux caractères Unicode équivalents, e.g. `html5['gt;'] == '>'`. À noter que le point-virgule en fin de chaîne est inclus dans le nom (e.g. `'gt;'`), toutefois certains noms sont acceptés par le standard même sans le point-virgule : dans ce cas, le nom est présent à la fois avec et sans le `;`. Voir aussi `html.unescape()`.

Nouveau dans la version 3.3.

`html.entities.entitydefs`

Un dictionnaire qui fait correspondre les définitions d'entités XHTML 1.0 avec leur remplacement en ISO Latin-1.

`html.entities.name2codepoint`

Un dictionnaire qui fait correspondre les noms d'entités HTML avec les points de code Unicode.

`html.entities.codepoint2name`

Un dictionnaire qui fait correspondre les points de code Unicode avec les noms d'entités HTML.

1. Voir <https://www.w3.org/TR/html5/syntax.html#named-character-references>

Notes

20.4 Modules de traitement XML

Code source : [Lib/xml/](#)

Les interfaces de Python de traitement de XML sont regroupées dans le paquet `xml`.

Avertissement : Les modules XML ne sont pas protégés contre les données mal construites ou malicieuses. Si vous devez parcourir des données douteuses non authentifiées voir les sections *Vulnérabilités XML* et *Les paquets defusedxml et defusedexpat*.

Il est important de noter que les modules dans le paquet `xml` nécessitent qu'au moins un analyseur compatible SAX soit disponible. L'analyseur Expat est inclus dans Python, ainsi le module `xml.parsers.expat` est toujours disponible.

La documentation des *bindings* des interfaces DOM et SAX se trouve dans `xml.dom` et `xml.sax`.

Les sous-modules de traitement XML sont :

- `xml.etree.ElementTree` : l'API ElementTree, un processeur simple et léger
- `xml.dom` : la définition de l'API DOM
- `xml.dom.minidom` : une implémentation minimale de DOM
- `xml.dom.pulldom` : gestion de la construction partiel des arbres DOM
- `xml.sax` : classes de bases SAX2 base et fonctions utilitaires
- `xml.parsers.expat` : le *binding* de l'analyseur Expat

20.4.1 Vulnérabilités XML

Les modules de traitement XML ne sont pas sécurisés contre les données construite malicieusement. Un attaquant peut abuser des fonctionnalités XML pour exécuter des attaques par déni de service, accéder des fichiers locaux, générer des connexions réseaux à d'autres machines ou contourner des pare-feux.

Le tableau suivant donne une vue d'ensemble des attaques connues et indique si les différents modules y sont vulnérables.

type	sax	etree	minidom	pulldom	xmlrpc
<i>billion laughs</i>	Vulnérable	Vulnérable	Vulnérable	Vulnérable	Vulnérable
<i>quadratic blowup</i>	Vulnérable	Vulnérable	Vulnérable	Vulnérable	Vulnérable
<i>external entity expansion</i>	Safe (4)	Sûr (1)	Sûr (2)	Safe (4)	Sûr (3)
Récupération de DTD	Safe (4)	Sûr	Sûr	Safe (4)	Sûr
<i>decompression bomb</i>	Sûr	Sûr	Sûr	Sûr	Vulnérable

1. `xml.etree.ElementTree` n'étend pas les entités externes et lève une exception `ParserError` quand une telle entité est rencontrée.
2. `xml.dom.minidom` n'étend pas les entités externe et renvoie simplement le verbatim de l'entité non étendu.
3. `xmlrpclib` n'étend pas les entités externes et les omet.
4. Since Python 3.8.0, external general entities are no longer processed by default since Python.

billion laughs / exponential entity expansion L'attaque **Billion Laughs** – aussi connue comme *exponential entity expansion* – utilise de multiples niveaux d'entités imbriquées. Chaque entité se réfère à une autre entité de multiple fois. L'entité finale contient une chaîne courte. Le résultat de l'expansion exponentielle génère plusieurs gigaoctet de texte et consomme beaucoup de mémoire et de temps processeur.

quadratic blowup entity expansion Une attaque *quadratic blowup* est similaire à l'attaque [Billion Laughs](#) ; il s'agit également d'un abus d'extension d'entités. Au lieu d'utiliser des entités imbriquées, cette attaque répète encore et encore une seule entité de plusieurs milliers de caractères. Cette attaque n'est pas aussi efficace que la version exponentielle mais contourne les contre-mesures de l'analyseur qui interdit les entités imbriquées de multiples fois.

external entity expansion Les déclarations d'entités peuvent contenir plus que du texte de substitution. Elles peuvent également référencer des ressources externes ou des fichiers locaux. L'analyseur XML accède à ces fichiers et inclut les contenus dans le document XML.

Récupération de DTD Certaines bibliothèques XML comme `xml.dom.pulldom` de Python récupère les documents de définitions de types (DTD) depuis des emplacements distants ou locaux. La fonctionnalité a des implications similaires que le problème d'extension d'entités externes.

decompression bomb Des bombes de décompression (ou **ZIP bomb**) sont valables pour toutes les bibliothèques XML qui peuvent analyser des flux XML compressés comme des flux HTTP *gzip* ou des fichiers compressés *LZMA*. Pour L'attaquant, cela permet de réduire d'une magnitude d'ordre 3 ou plus la quantité de données transmises.

La documentation de [defusedxml](#) sur PyPI contient plus d'informations sur tous les vecteurs d'attaques connus ainsi que des exemples et des références.

20.4.2 Les paquets `defusedxml` et `defusedexpat`

`defusedxml` est un paquet écrit exclusivement en Python avec des sous-classe modifiées de tous les analyseurs de la *stdlib* XML qui empêche toutes opérations potentiellement malicieuses. L'utilisation de ce paquet est recommandé pour tous serveurs qui analyseraient des données XML non fiables. Le paquet inclut également des exemples d'attaques et une documentation plus fournie sur plus d'attaques XML comme *XPath injection*.

`defusedexpat` fournit une version modifiée de *libexpat* et le module `pyexpat` modifiée embarquant des contre-mesures contre les attaques *DoS* par *entity expansion*. Le module `defusedexpat` autorise un nombre configurable et raisonnable d'extension d'entités. Ces modifications pourraient être incluses dans des futures version de Python mais ne seront incluses dans aucune version corrective de Python pour éviter de casser la compatibilité rétrograde.

20.5 `xml.etree.ElementTree` — The `ElementTree` XML API

Source code : [Lib/xml/etree/ElementTree.py](#)

The `xml.etree.ElementTree` module implements a simple and efficient API for parsing and creating XML data.

Modifié dans la version 3.3 : This module will use a fast implementation whenever available. The `xml.etree.cElementTree` module is deprecated.

Avertissement : The `xml.etree.ElementTree` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [Vulnérabilités XML](#).

20.5.1 Tutoriel

This is a short tutorial for using `xml.etree.ElementTree` (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose - `ElementTree` represents the whole XML document as a tree, and `Element` represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the `ElementTree` level. Interactions with a single XML element and its sub-elements are done on the `Element` level.

Parsing XML

We'll be using the following XML document as the sample data for this section :

```
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank>1</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank>4</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
  <country name="Panama">
    <rank>68</rank>
    <year>2011</year>
    <gdppc>13600</gdppc>
    <neighbor name="Costa Rica" direction="W"/>
    <neighbor name="Colombia" direction="E"/>
  </country>
</data>
```

We can import this data by reading from a file :

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Or directly from a string :

```
root = ET.fromstring(country_data_as_string)
```

`fromstring()` parses XML from a string directly into an `Element`, which is the root element of the parsed tree. Other parsing functions may create an `ElementTree`. Check the documentation to be sure.

As an `Element`, `root` has a tag and a dictionary of attributes :

```
>>> root.tag
'data'
>>> root.attrib
{}
```

It also has children nodes over which we can iterate :

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Children are nested, and we can access specific child nodes by index :

```
>>> root[0][1].text
'2008'
```

Note : Not all elements of the XML input will end up as elements of the parsed tree. Currently, this module skips over any XML comments, processing instructions, and document type declarations in the input. Nevertheless, trees built using this module's API rather than parsing from XML text can have comments and processing instructions in them ; they will be included when generating XML output. A document type declaration may be accessed by passing a custom *TreeBuilder* instance to the *XMLParser* constructor.

Pull API for non-blocking parsing

Most parsing functions provided by this module require the whole document to be read at once before returning any result. It is possible to use an *XMLParser* and feed data into it incrementally, but it is a push API that calls methods on a callback target, which is too low-level and inconvenient for most needs. Sometimes what the user really wants is to be able to parse XML incrementally, without blocking operations, while enjoying the convenience of fully constructed *Element* objects.

The most powerful tool for doing this is *XMLPullParser*. It does not require a blocking read to obtain the XML data, and is instead fed with data incrementally with *XMLPullParser.feed()* calls. To get the parsed XML elements, call *XMLPullParser.read_events()*. Here is an example :

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
```

The obvious use case is applications that operate in a non-blocking fashion where the XML data is being received from a socket or read incrementally from some storage device. In such cases, blocking reads are unacceptable.

Because it's so flexible, *XMLPullParser* can be inconvenient to use for simpler use-cases. If you don't mind your application blocking on reading XML data but would still like to have incremental parsing capabilities, take a look at *iterparse()*. It can be useful when you're reading a large XML document and don't want to hold it wholly in memory.

Finding interesting elements

Element has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, *Element.iter()* :

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() finds only elements with a tag which are direct children of the current element. *Element.find()* finds the *first* child with a particular tag, and *Element.text* accesses the element's text content. *Element.get()* accesses the element's attributes :

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using *XPath*.

Modifying an XML File

ElementTree provides a simple way to build XML documents and write them to files. The *ElementTree.write()* method serves this purpose.

Once created, an *Element* object may be manipulated by directly changing its fields (such as *Element.text*), adding and modifying attributes (*Element.set()* method), as well as adding new children (for example with *Element.append()*).

Let's say we want to add one to each country's rank, and add an updated attribute to the rank element :

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

Our XML now looks like this :

```
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank updated="yes">2</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
```

(suite sur la page suivante)

(suite de la page précédente)

```

</country>
<country name="Singapore">
  <rank updated="yes">5</rank>
  <year>2011</year>
  <gdppc>59900</gdppc>
  <neighbor name="Malaysia" direction="N"/>
</country>
<country name="Panama">
  <rank updated="yes">69</rank>
  <year>2011</year>
  <gdppc>13600</gdppc>
  <neighbor name="Costa Rica" direction="W"/>
  <neighbor name="Colombia" direction="E"/>
</country>
</data>

```

We can remove elements using `Element.remove()`. Let's say we want to remove all countries with a rank higher than 50 :

```

>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')

```

Our XML now looks like this :

```

<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank updated="yes">2</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank updated="yes">5</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
</data>

```

Building XML documents

The `SubElement()` function also provides a convenient way to create new sub-elements for a given element :

```

>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>

```


Parsing XML with Namespaces

If the XML input has `namespaces`, tags and attributes with prefixes in the form `prefix:sometag` get expanded to `{uri}sometag` where the *prefix* is replaced by the full *URI*. Also, if there is a `default namespace`, that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix « fictional » and the other serving as the default namespace :

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
  <actor>
    <name>John Cleese</name>
    <fictional:character>Lancelot</fictional:character>
    <fictional:character>Archie Leach</fictional:character>
  </actor>
  <actor>
    <name>Eric Idle</name>
    <fictional:character>Sir Robin</fictional:character>
    <fictional:character>Gunther</fictional:character>
    <fictional:character>Commander Clement</fictional:character>
  </actor>
</actors>
```

One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a `find()` or `findall()` :

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions :

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

These two approaches both output :

```
John Cleese
|--> Lancelot
|--> Archie Leach
Eric Idle
|--> Sir Robin
|--> Gunther
|--> Commander Clement
```

Additional resources

See <http://effbot.org/zone/element-index.htm> for tutorials and links to other docs.

20.5.2 XPath support

This module provides limited support for *XPath expressions* for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

Exemple

Here's an example that demonstrates some of the XPath capabilities of the module. We'll be using the `countrydata` XML document from the *Parsing XML* section :

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall("./year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall("./*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall("./neighbor[2]")
```

Supported XPath syntax

Syntaxe	Signification
<code>tag</code>	Selects all child elements with the given tag. For example, <code>spam</code> selects all child elements named <code>spam</code> , and <code>spam/egg</code> selects all grandchildren named <code>egg</code> in all children named <code>spam</code> .
<code>*</code>	Selects all child elements. For example, <code>*/egg</code> selects all grandchildren named <code>egg</code> .
<code>.</code>	Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.
<code>//</code>	Selects all subelements, on all levels beneath the current element. For example, <code>./egg</code> selects all <code>egg</code> elements in the entire tree.
<code>..</code>	Selects the parent element. Returns <code>None</code> if the path attempts to reach the ancestors of the start element (the element <code>find</code> was called on).
<code>[@attrib]</code>	Selects all elements that have the given attribute.
<code>[@attrib='value']</code>	Selects all elements for which the given attribute has the given value. The value cannot contain quotes.
<code>[tag]</code>	Selects all elements that have a child named <code>tag</code> . Only immediate children are supported.
<code>[tag='text']</code>	Selects all elements that have a child named <code>tag</code> whose complete text content, including descendants, equals the given <code>text</code> .
<code>[position]</code>	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression <code>last()</code> (for the last position), or a position relative to the last position (e.g. <code>last()-1</code>).

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. `position` predicates must be preceded by a tag name.

20.5.3 Référence

Fonctions

`xml.etree.ElementTree.Comment` (*text=None*)

Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. *text* is a string containing the comment string. Returns an element instance representing a comment.

Note that *XMLParser* skips over comments in the input instead of creating comment objects for them. An *ElementTree* will only contain comment nodes if they have been inserted into the tree using one of the *Element* methods.

`xml.etree.ElementTree.dump` (*elem*)

Writes an element tree or element structure to `sys.stdout`. This function should be used for debugging only. The exact output format is implementation dependent. In this version, it's written as an ordinary XML file. *elem* is an element tree or an individual element.

`xml.etree.ElementTree.fromstring` (*text*)

Parses an XML section from a string constant. Same as *XML()*. *text* is a string containing XML data. Returns an *Element* instance.

`xml.etree.ElementTree.fromstringlist` (*sequence, parser=None*)

Parses an XML document from a sequence of string fragments. *sequence* is a list or other sequence containing XML data fragments. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *Element* instance.

Nouveau dans la version 3.2.

`xml.etree.ElementTree.iselement(element)`

Checks if an object appears to be a valid element object. *element* is an element instance. Returns a true value if this is an element object.

`xml.etree.ElementTree.iterparse(source, events=None, parser=None)`

Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a filename or *file object* containing XML data. *events* is a sequence of events to report back. The supported events are the strings "start", "end", "start-ns" and "end-ns" (the « ns » events are used to get detailed namespace information). If *events* is omitted, only "end" events are reported. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. *parser* must be a subclass of *XMLParser* and can only use the default *TreeBuilder* as a target. Returns an *iterator* providing (*event*, *elem*) pairs.

Note that while *iterparse()* builds the tree incrementally, it issues blocking reads on *source* (or the file it names). As such, it's unsuitable for applications where blocking reads can't be made. For fully non-blocking parsing, see *XMLPullParser*.

Note : *iterparse()* only guarantees that it has seen the « > » character of a starting tag when it emits a « start » event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for « end » events instead.

Obsolète depuis la version 3.4 : The *parser* argument.

`xml.etree.ElementTree.parse(source, parser=None)`

Parses an XML section into an element tree. *source* is a filename or file object containing XML data. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *ElementTree* instance.

`xml.etree.ElementTree.ProcessingInstruction(target, text=None)`

PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. *target* is a string containing the PI target. *text* is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

Note that *XMLParser* skips over processing instructions in the input instead of creating comment objects for them. An *ElementTree* will only contain processing instruction nodes if they have been inserted into to the tree using one of the *Element* methods.

`xml.etree.ElementTree.register_namespace(prefix, uri)`

Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. *prefix* is a namespace prefix. *uri* is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

Nouveau dans la version 3.2.

`xml.etree.ElementTree.SubElement(parent, tag, attrib={}, **extra)`

Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *parent* is the parent element. *tag* is the subelement name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments. Returns an element instance.

`xml.etree.ElementTree.tostring(element, encoding="us-ascii", method="xml", *,
 short_empty_elements=True)`

Generates a string representation of an XML element, including all subelements. *element* is an *Element* instance. *encoding*¹ is the output encoding (default is US-ASCII). Use *encoding*="unicode" to generate a Unicode string (otherwise, a bytestring is generated). *method* is either "xml", "html" or "text" (default is "xml").

1. The encoding string included in XML output should conform to the appropriate standards. For example, « UTF-8 » is valid, but « UTF8 » is not. See <https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl> and <https://www.iana.org/assignments/character-sets/character-sets.xhtml>.

`short_empty_elements` has the same meaning as in `ElementTree.write()`. Returns an (optionally) encoded string containing the XML data.

Nouveau dans la version 3.4 : Le paramètre `short_empty_elements`.

```
xml.etree.ElementTree.tostringlist(element, encoding="us-ascii", method="xml", *,
                                   short_empty_elements=True)
```

Generates a string representation of an XML element, including all subelements. `element` is an `Element` instance. `encoding`¹ is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). `method` is either "xml", "html" or "text" (default is "xml"). `short_empty_elements` has the same meaning as in `ElementTree.write()`. Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that `b""`. `join(tostringlist(element)) == tostring(element)`.

Nouveau dans la version 3.2.

Nouveau dans la version 3.4 : Le paramètre `short_empty_elements`.

```
xml.etree.ElementTree.XML(text, parser=None)
```

Parses an XML section from a string constant. This function can be used to embed « XML literals » in Python code. `text` is a string containing XML data. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

```
xml.etree.ElementTree.XMLID(text, parser=None)
```

Parses an XML section from a string constant, and also returns a dictionary which maps from element id :s to elements. `text` is a string containing XML data. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns a tuple containing an `Element` instance and a dictionary.

Objets Elements

```
class xml.etree.ElementTree.Element(tag, attrib={}, **extra)
```

Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. `tag` is the element name. `attrib` is an optional dictionary, containing element attributes. `extra` contains additional attributes, given as keyword arguments.

tag

A string identifying what kind of data this element represents (the element type, in other words).

text

tail

These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the `text` attribute holds either the text between the element's start tag and its first child or end tag, or `None`, and the `tail` attribute holds either the text between the element's end tag and the next tag, or `None`. For the XML data

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

the `a` element has `None` for both `text` and `tail` attributes, the `b` element has `text` "1" and `tail` "4", the `c` element has `text` "2" and `tail` `None`, and the `d` element has `text` `None` and `tail` "3".

To collect the inner text of an element, see `itertext()`, for example `"".join(element.itertext())`.

Applications may store arbitrary objects in these attributes.

attrib

A dictionary containing the element's attributes. Note that while the `attrib` value is always a real mutable Python dictionary, an `ElementTree` implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

clear()

Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to `None`.

get (*key*, *default=None*)

Gets the element attribute named *key*.

Returns the attribute value, or *default* if the attribute was not found.

items()

Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

keys()

Returns the elements attribute names as a list. The names are returned in an arbitrary order.

set (*key*, *value*)

Set the attribute *key* on the element to *value*.

The following methods work on the element's children (subelements).

append (*subelement*)

Adds the element *subelement* to the end of this element's internal list of subelements. Raises `TypeError` if *subelement* is not an `Element`.

extend (*subelements*)

Appends *subelements* from a sequence object with zero or more elements. Raises `TypeError` if a subelement is not an `Element`.

Nouveau dans la version 3.2.

find (*match*, *namespaces=None*)

Finds the first subelement matching *match*. *match* may be a tag name or a *path*. Returns an element instance or `None`. *namespaces* is an optional mapping from namespace prefix to full name.

findall (*match*, *namespaces=None*)

Finds all matching subelements, by tag name or *path*. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

findtext (*match*, *default=None*, *namespaces=None*)

Finds text for the first subelement matching *match*. *match* may be a tag name or a *path*. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned. *namespaces* is an optional mapping from namespace prefix to full name.

getchildren()

Obsolète depuis la version 3.2 : Use `list(elem)` or iteration.

getiterator (*tag=None*)

Obsolète depuis la version 3.2 : Use method `Element.iter()` instead.

insert (*index*, *subelement*)

Inserts *subelement* at the given position in this element. Raises `TypeError` if *subelement* is not an `Element`.

iter (*tag=None*)

Creates a tree *iterator* with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If *tag* is not `None` or `'*'`, only elements whose tag equals *tag* are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

Nouveau dans la version 3.2.

iterfind (*match*, *namespaces=None*)

Finds all matching subelements, by tag name or *path*. Returns an iterable yielding all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

Nouveau dans la version 3.2.

itertext()

Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

Nouveau dans la version 3.2.

makeelement (*tag, attrib*)

Creates a new element object of the same type as this element. Do not call this method, use the `SubElement()` factory function instead.

remove (*subelement*)

Removes *subelement* from the element. Unlike the `find*` methods this method compares elements based on the instance identity, not on tag value or contents.

`Element` objects also support the following sequence type methods for working with subelements : `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`.

Caution : Elements with no subelements will test as `False`. This behavior will change in future versions. Use specific `len(elem)` or `elem is None` test instead.

```
element = root.find('foo')

if not element: # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

ElementTree Objects

class `xml.etree.ElementTree.ElementTree` (*element=None, file=None*)

`ElementTree` wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

element is the root element. The tree is initialized with the contents of the XML *file* if given.

_setroot (*element*)

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. *element* is an element instance.

find (*match, namespaces=None*)

Same as `Element.find()`, starting at the root of the tree.

findall (*match, namespaces=None*)

Same as `Element.findall()`, starting at the root of the tree.

findtext (*match, default=None, namespaces=None*)

Same as `Element.findtext()`, starting at the root of the tree.

getiterator (*tag=None*)

Obsolète depuis la version 3.2 : Use method `ElementTree.iter()` instead.

getroot ()

Returns the root element for this tree.

iter (*tag=None*)

Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. *tag* is the tag to look for (default is to return all elements).

iterfind (*match, namespaces=None*)

Same as `Element.iterfind()`, starting at the root of the tree.

Nouveau dans la version 3.2.

parse (*source, parser=None*)

Loads an external XML section into this element tree. *source* is a file name or *file object*. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns the section root element.

write (*file, encoding="us-ascii", xml_declaration=None, default_namespace=None, method="xml", *, short_empty_elements=True*)

Writes the element tree to a file, as XML. *file* is a file name, or a *file object* opened for writing. *encoding*¹ is the output encoding (default is US-ASCII). *xml_declaration* controls if an XML declaration should be added to the file. Use `False` for never, `True` for always, `None` for only if not US-ASCII or UTF-8 or

Unicode (default is `None`). *default_namespace* sets the default XML namespace (for « `xmlns` »). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). The keyword-only *short_empty_elements* parameter controls the formatting of elements that contain no content. If `True` (the default), they are emitted as a single self-closed tag, otherwise they are emitted as a pair of start/end tags.

The output is either a string (*str*) or binary (*bytes*). This is controlled by the *encoding* argument. If *encoding* is `"unicode"`, the output is a string; otherwise, it's binary. Note that this may conflict with the type of *file* if it's an open *file object*; make sure you do not try to write a string to a binary stream and vice versa.

Nouveau dans la version 3.4 : Le paramètre *short_empty_elements*.

This is the XML file that is going to be manipulated :

```
<html>
  <head>
    <title>Example page</title>
  </head>
  <body>
    <p>Moved to <a href="http://example.org/">example.org</a>
    or <a href="http://example.com/">example.com</a>.</p>
  </body>
</html>
```

Example of changing the attribute « `target` » of every link in first paragraph :

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")      # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))    # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:              # Iterates through all found links
...     i.attrib["target"] = "blank"
>>> tree.write("output.xhtml")
```

QName Objects

class `xml.etree.ElementTree.QName` (*text_or_uri*, *tag=None*)

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. *text_or_uri* is a string containing the QName value, in the form `{uri}local`, or, if the *tag* argument is given, the URI part of a QName. If *tag* is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. *QName* instances are opaque.

TreeBuilder Objects

class `xml.etree.ElementTree.TreeBuilder` (*element_factory=None*)

Generic element structure builder. This builder converts a sequence of start, data, and end method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. *element_factory*, when given, must be a callable accepting two positional arguments : a tag and a dict of attributes. It is expected to return a new element instance.

close ()

Flushes the builder buffers, and returns the toplevel document element. Returns an *Element* instance.

data (*data*)

Adds text to the current element. *data* is a string. This should be either a bytestring, or a Unicode string.

end (*tag*)

Closes the current element. *tag* is the element name. Returns the closed element.

start (*tag, attrs*)

Opens a new element. *tag* is the element name. *attrs* is a dictionary containing element attributes. Returns the opened element.

In addition, a custom *TreeBuilder* object can provide the following method :

doctype (*name, pubid, system*)

Handles a doctype declaration. *name* is the doctype name. *pubid* is the public identifier. *system* is the system identifier. This method does not exist on the default *TreeBuilder* class.

Nouveau dans la version 3.2.

XMLParser Objects

class `xml.etree.ElementTree.XMLParser` (*html=0, target=None, encoding=None*)

This class is the low-level building block of the module. It uses `xml.parsers.expat` for efficient, event-based parsing of XML. It can be fed XML data incrementally with the `feed()` method, and parsing events are translated to a push API - by invoking callbacks on the *target* object. If *target* is omitted, the standard *TreeBuilder* is used. The *html* argument was historically used for backwards compatibility and is now deprecated. If *encoding*¹ is given, the value overrides the encoding specified in the XML file.

Obsolète depuis la version 3.4 : The *html* argument. The remaining arguments should be passed via keyword to prepare for the removal of the *html* argument.

close ()

Finishes feeding data to the parser. Returns the result of calling the `close()` method of the *target* passed during construction ; by default, this is the toplevel document element.

doctype (*name, pubid, system*)

Obsolète depuis la version 3.2 : Define the `TreeBuilder.doctype()` method on a custom *TreeBuilder* target.

feed (*data*)

Feeds data to the parser. *data* is encoded data.

`XMLParser.feed()` calls *target's* `start(tag, attrs_dict)` method for each opening tag, its `end(tag)` method for each closing tag, and data is processed by method `data(data)`. `XMLParser.close()` calls *target's* method `close()`. `XMLParser` can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file :

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                                # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):                # Called for each opening tag.
...         self.depth += 1
```

(suite sur la page suivante)

(suite de la page précédente)

```

...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):          # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass                    # We do not need to do anything with data.
...     def close(self):            # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...     </b>
...     <b>
...       <c>
...         <d>
...           </d>
...         </c>
...       </b>
...     </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4

```

XMLPullParser Objects

class xml.etree.ElementTree.XMLPullParser (*events=None*)

A pull parser suitable for non-blocking applications. Its input-side API is similar to that of *XMLParser*, but instead of pushing calls to a callback target, *XMLPullParser* collects an internal list of parsing events and lets the user read from it. *events* is a sequence of events to report back. The supported events are the strings "start", "end", "start-ns" and "end-ns" (the « ns » events are used to get detailed namespace information). If *events* is omitted, only "end" events are reported.

feed (*data*)

Feed the given bytes data to the parser.

close ()

Signal the parser that the data stream is terminated. Unlike *XMLParser.close()*, this method always returns *None*. Any events not yet retrieved when the parser is closed can still be read with *read_events()*.

read_events ()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields (*event*, *elem*) pairs, where *event* is a string representing the type of event (e.g. "end") and *elem* is the encountered *Element* object.

Events provided in a previous call to *read_events()* will not be yielded again. Events are consumed from the internal queue only when they are retrieved from the iterator, so multiple readers iterating in parallel over iterators obtained from *read_events()* will have unpredictable results.

Note : *XMLPullParser* only guarantees that it has seen the « > » character of a starting tag when it emits a « start » event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for « end » events instead.

Nouveau dans la version 3.4.

Exceptions

class `xml.etree.ElementTree.ParseError`

XML parse error, raised by the various parsing methods in this module when parsing fails. The string representation of an instance of this exception will contain a user-friendly error message. In addition, it will have the following attributes available :

code

A numeric error code from the expat parser. See the documentation of `xml.parsers.expat` for the list of error codes and their meanings.

position

A tuple of *line*, *column* numbers, specifying where the error occurred.

Notes

20.6 `xml.dom` — L'API Document Object Model

Code source : `Lib/xml/dom/__init__.py`

Le Document Object Model, ou « DOM, » est une API inter-langage du World Wide Web Consortium (W3C) pour accéder et modifier les documents XML. Une implémentation DOM présente le document XML comme un arbre ou autorise le code client à construire une telle structure depuis zéro. Il permet alors d'accéder à la structure à l'aide d'un ensemble d'objet qui fournissent des interfaces bien connues.

Le DOM est extrêmement utile pour les applications à accès aléatoire. SAX ne vous permet de visualiser qu'un seul morceau du document à la fois. Si vous regardez un élément SAX, vous n'avez pas accès à un autre. Si vous regardez un nœud de texte, vous n'avez pas accès à un élément parent. Lorsque vous écrivez une application SAX, vous devez suivre la position de votre programme dans le document quelque part dans votre propre code. SAX ne le fait pas pour vous. De plus, si vous devez examiner un nœud plus loin dans le document XML, vous n'avez pas de chance.

Il est tout simplement impossible d'implémenter certains algorithmes avec un modèle évènementiel, sans un accès à un arbre. Bien sûr, vous pourriez construire vous même un arbre à partir des événements SAX mais DOM vous permet d'éviter d'écrire ce code. Le DOM est représentation standard en arbre pour des données XML.

Le DOM (Document Object Model) est défini par le W3C en étapes ou « *levels* » (niveaux) selon leur terminologie. Le couplage de l'API de Python est essentiellement basée sur la recommandation DOM Level 2.

Typiquement, les applications DOM commencent par analyser du XML dans du DOM. Comment cela doit être exposé n'est absolument pas décrit par DOM Level 1 et Level 2 ne fournit que des améliorations limitées. Il existe une classe `DOMImplementation` qui fournit un accès à des méthodes de création de `Document` mais il n'y a pas de moyen d'accéder à un lecteur/analyseur/constructeur de *document* de façon indépendante de l'implémentation. Il n'est pas également très bien définis comment accéder à ces méthodes sans un objet `Document`. En Python, chaque implémentation fournira une fonction `getDOMImplementation()`. DOM Level 3 ajoute une spécification *Load/Store* (charge/stocke) qui définit une interface pour le lecteur mais qui n'est pas disponible dans la bibliothèque standard de Python.

Une fois que vous avez un objet document DOM, vous pouvez accéder aux parties de votre document XML à travers ses méthodes et propriétés. Ces propriétés sont définies dans les spécifications DOM ; cette portion du manuel de références décrit l'interprétation des ces spécifications en Python.

Les spécifications fournies par le W3C définissent les API DOM pour Java, ECMAScript, et OMG IDL. Les correspondances de Python définies ici sont basées pour une grande part sur la version IDL de la spécification mais une conformité stricte n'est pas requise (bien que ces implémentations soient libre d'implémenter le support strict des correspondances de IDL). Voir la section *Conformité* pour une discussion détaillée des pré-requis des correspondances.

Voir aussi :

Spécification Level 2 Document Object Model (DOM) La recommandation W3C sur laquelle l'API DOM de Python est basée.

Spécification Level 1 Document Object Model (DOM) La recommandation du W3C pour le DOM supporté par `xml.dom.minidom`.

Python Language Mapping Specification Ceci spécifie les correspondances depuis OMG IDL vers Python.

20.6.1 Contenu du module

Le `xml.dom` contient les fonctions suivantes :

`xml.dom.registerDOMImplementation(name, factory)`

Enregistre la fonction *factory* avec le nom *name*. La fonction *factory* doit renvoyer un objet qui implémente l'interface de `DOMImplementation`. La fonction *factory* peut renvoyer le même objet à chaque fois ou un nouveau à chaque appel en accord avec les spécificités de l'implémentation (Par exemple si l'implémentation supporte certaines personnalisations).

`xml.dom.getDOMImplementation(name=None, features=())`

Renvoie une implémentation DOM appropriée. Le *name* est soit connu, soit le nom du module d'une implémentation DOM, soit `None`. Si ce n'est pas `None`, le module correspondant est importé et retourne un objet `DOMImplementation` si l'importation réussit. Si Aucun *name* n'est donné et que la variable d'environnement `PYTHON_DOM` est positionnée, cette variable est utilisée pour trouver l'implémentation.

Si *name* n'est pas donné, la fonction examine les implémentations disponibles pour en trouver une avec l'ensemble des fonctionnalités requises. Si aucune implémentation n'est trouvée, une `ImportError` est levée. La liste de fonctionnalité doit être une séquence de paires (*feature*, *version*) qui est passée à la méthode `hasFeature()` disponible dans les objets `DOMImplementation`.

Quelques constantes pratiques sont également fournies :

`xml.dom.EMPTY_NAMESPACE`

La valeur utilisée pour indiquer qu'aucun espace de noms n'est associé à un nœud dans le DOM. Typiquement, ceci est trouvé comme `namespaceURI` dans un nœud ou utilisé comme le paramètre *namespaceURI* dans une méthode spécifique aux espaces de noms.

`xml.dom.XML_NAMESPACE`

L'URI de l'espace de noms associé avec le préfixe réservé `xml` comme défini par [Namespaces in XML](#) (section 4).

`xml.dom.XMLNS_NAMESPACE`

L'URI de l'espace de noms pour la déclaration des espaces de noms, tel que défini par [Document Object Model \(DOM\) Level 2 Core Specification](#) (section 1.1.8).

`xml.dom.XHTML_NAMESPACE`

L'URI de l'espace de noms XHTML tel que défini par [XHTML 1.0 : The Extensible HyperText Markup Language](#) (section 3.1.1).

Par ailleurs, `xml.dom` contient une classe de base `Node` et les exceptions de DOM. La classe `Node` fournie par ce module n'implémente aucune des méthodes ou des attributs définis par les spécifications DOM ; les implémentations concrètes des DOM doivent fournir les informations suivantes. La classe `Node` fournie par ce module fournit les constantes utilisées pour l'attribut `nodeType` pour des objets concrets `Node` ; ils sont situés dans les classes plutôt qu'au niveau du module en accord avec les spécifications DOM.

20.6.2 Objets dans le DOM

La documentation finale pour le DOM est la spécification DOM du W3C.

Notez que les attributs DOM peuvent également être manipulés comme des nœuds au lieu de simples chaînes. Il est relativement rare que vous ayez besoin de faire cela, cependant, cet usage n'est pas encore documenté.

Interface	Section	Objectif
DOMImplementation	<i>Objets DOMImplementation</i>	Interface de l'implémentation sous-jacente.
Node	<i>Objets nœuds</i>	Interface de base pour la majorité des objets dans un document.
NodeList	<i>Objet NodeList</i>	Interface pour une séquence de nœuds.
DocumentType	<i>Objets DocumentType</i>	Informations sur les déclarations nécessaires au traitement d'un document.
Document	<i>Objets Document</i>	Objet représentant un document entier.
Element	<i>Objets Elements</i>	Nœuds éléments dans la hiérarchie d'un document.
Attr	<i>Objets Attr</i>	Valeur des nœuds attributs sur dans des nœuds éléments.
Comment	<i>Objets Comment</i>	Représentation des commentaires dans le fichier source du document.
Text	<i>Objets Text et CDATA-Section</i>	Nœud contenant un contenu texte du document.
ProcessingInstruction	<i>Objets ProcessingInstruction</i>	Représentation des <i>Processing Instructions</i> .

Une Section additionnelle décrit les exceptions définis pour travailler avec le DOM en Python.

Objets DOMImplementation

L'interface `DOMImplementation` fournit un moyen pour les applications de déterminer la disponibilité de fonctionnalités particulières dans le DOM qu'elles utilisent. *DOM Level 2* ajoute la capacité de créer des nouveaux objets `Document` et `DocumentType` utilisant également `DOMImplementation`.

`DOMImplementation.hasFeature` (*feature*, *version*)

Renvoie vrai si la fonctionnalité identifiée par une paire de chaîne *feature* et *version* est implémentée.

`DOMImplementation.createDocument` (*namespaceUri*, *qualifiedName*, *doctype*)

Renvoie un nouvel objet `Document` (la racine du DOM), avec un objet fils `Element` ayant les *namespaceUri* et *qualifiedName* passés en paramètre. Le *doctype* doit être un objet `DocumentType` créé par `createDocumentType()` ou `None`. Dans l'API DOM de Python, les deux premiers arguments peuvent également être à `None` de manière à indiquer qu'aucun enfant `Element` ne soit créé.

`DOMImplementation.createDocumentType` (*qualifiedName*, *publicId*, *systemId*)

Renvoie un nouvel objet `DocumentType` qui encapsule les chaînes *qualifiedName*, *publicId*, et *systemId* passées en paramètre représentant les informations contenues dans la déclaration du document XML.

Objets nœuds

Tous les composants d'un document XML sont des sous-classes de `Node`.

`Node.nodeType`

Un entier représentant le type de nœud. Pour l'objet `Node`, les constantes symboliques pour les types sont `ELEMENT_NODE`, `ATTRIBUTE_NODE`, `TEXT_NODE`, `CDATA_SECTION_NODE`, `ENTITY_NODE`, `PROCESSING_INSTRUCTION_NODE`, `COMMENT_NODE`, `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `NOTATION_NODE`. Ceci est un attribut en lecture seule.

`Node.parentNode`

Le parent du nœud courant ou `None` dans le cas du nœud document. La valeur est toujours un objet `Node` ou `None`. Pour les nœuds `Element`, ce sera le parent de l'élément sauf si l'élément est la racine, dans ce cas ce sera l'objet `Document`. Pour les nœuds `Attr`, cela sera toujours `None`. Ceci est un attribut en lecture seule.

`Node.attributes`

Un objet `NamedNodeMap` d'objet attributs. Seulement les éléments ayant des valeurs seront listés, les autres renverront `None` pour cet attribut. Cet attribut est en lecture seule.

`Node.previousSibling`

Le nœud avec le même parent qui précède immédiatement le nœud courant. Par exemple, l'élément avec la balise `fermente` qui est juste avant la balise ouvrante de l'élément *self*. Naturellement, les documents XML sont fait de plus que juste des éléments ; donc le *previous sibling* peut être du texte, un commentaire ou autre chose. Si le nœud courant est le premier fils du parent, cet attribut vaudra `None`. Cet attribut est en lecture seule.

`Node.nextSibling`

Le nœud qui suit immédiatement le nœud courant dans le même parent. Voir également [previousSibling](#). Si ce nœud est le dernier de son parent, alors l'attribut sera `None`. Cet attribut est en lecture seule.

`Node.childNodes`

Une liste de nœuds contenu dans le nœud courant. Cet attribut est en lecture seule.

`Node.firstChild`

S'il y a des fils, premier fils du nœud courant, sinon `None`. Cet attribut est en lecture seule.

`Node.lastChild`

S'il y a des fils, le dernier nœud fils du nœud courant. Sinon `None`. Cet attribut est en lecture seule.

`Node.localName`

S'il y a un `:`, contient la partie suivante de `tagName` ce `:` sinon la valeur complète de `tagName`. Cette valeur est une chaîne.

`Node.prefix`

La partie de `tagName` précédent le `:` s'il y en a un, sinon une chaîne vide. La valeur est une chaîne ou `None`.

`Node.namespaceURI`

L'espace de noms associé (*namespace* en anglais) au nom de l'élément. Cette valeur est une chaîne ou `None`. Cet attribut est en lecture seule.

`Node.nodeName`

L'attribut a un sens différent pour chaque type de nœud ; se reporter à la spécification DOM pour les détails. Vous obtiendrez toujours l'information que vous obtiendrez à l'aide d'une autre propriété comme `tagName` pour les éléments ou `name` pour les attributs. Pour tous les types de nœuds, la valeur sera soit une chaîne soit `None`. Cet attribut est en lecture seule.

`Node.nodeValue`

L'attribut a un sens différent pour chaque type de nœud ; se reporter à la spécification DOM pour les détails. La situation est similaire à [nodeName](#). La valeur est une chaîne ou `None`.

`Node.hasAttributes()`

Renvoi vrai si le nœud a des attributs.

`Node.hasChildNodes()`

Renvoie vrai si le nœud a des nœuds fils.

`Node.isSameNode(other)`

Renvoie vrai si *other* fait référence au même nœud que le nœud courant. Ceci est particulièrement pratique pour l'implémentation de DOM qui utilise une architecture proxy (car plus d'un objet peut se référer au même nœud).

Note : Ceci est basé sur l'API proposé par * DOM Level 3* qui est toujours à l'étape « *working draft* » mais cette interface particulière ne paraît pas controversée. Les changements de W3C n'affecteront pas nécessairement cette méthode dans l'interface DOM de Python. (bien que toute nouvelle API W3C à cet effet soit également supportée).

`Node.appendChild(newChild)`

Ajoute un nouveau nœud fils à ce nœud à la fin de la liste des fils renvoyant *newChild*. Si ce nœud est déjà dans l'arbre, il sera d'abord retiré.

`Node.insertBefore(newChild, refChild)`

Insère un nouveau nœud fils avant un fils existant. Il est impératif que *refChild* soit un fils du nœud, sinon *ValueError* sera levée. *newChild* est renvoyé. Si *refChild* est *None*, *newChild* est inséré à la fin de la liste des fils.

`Node.removeChild(oldChild)`

Retire un nœud fils. *oldChild* doit être un fils de ce nœud ; sinon *ValueError* sera levée. En cas de succès, *oldChild* est renvoyé. Si *oldChild* n'est plus utilisé, sa méthode *unlink()* doit être appelée.

`Node.replaceChild(newChild, oldChild)`

Remplace un nœud existant avec un nouveau. *oldChild* doit être un fils de ce nœud ; sinon *ValueError* sera levée.

`Node.normalize()`

Jointe les nœuds texte adjacents de manière à ce que tous les segments de texte soient stockés dans une seule instance de *Text*. Ceci simplifie le traitement du texte d'un arbre DOM pour de nombreuses applications.

`Node.cloneNode(deep)`

Clone ce nœud. Positionner *deep* signifie que tous les nœuds fils seront également clonés. La méthode renvoie le clone.

Objet NodeList

NodeList représente une séquence de nœuds. Ces objets sont utilisés de deux manières dans la recommandation Dom Core : un objet *Element* fournit en fournissant la liste des nœuds fils et les méthodes *getElementsByTagName()* et *getElementsByTagNameNS()* de *Node* renvoient des objets avec cette interface pour représenter les résultats des requêtes.

La recommandation DOM Level 2 définit un attribut et une méthode pour ces objets :

NodeList.item(i)

Renvoie le *i**ème élément de la séquence s'il existe ou *None*. L'index *i* ne peut pas être inférieur à 0 ou supérieur ou égale à la longueur de la séquence.

NodeList.length

Le nombre d'éléments dans la séquence.

En plus, l'interface DOM de Python requiert quelques ajouts supplémentaires pour permettre que les objets *NodeList* puissent être utilisés comme des séquences Python. Toutes les implémentations de *NodeList* doivent inclure le support de *__len__()* et de *__getitem__()* ; ceci permet l'itération sur *NodeList* avec l'instruction *for* et un support de la fonction native *len()*.

Si une implémentation de DOM supporte les modifications du document, l'implémentation de *NodeList* doit également supporter les méthodes *__setitem__()* et *__delitem__()*.

Objets DocumentType

Les objets de type `DocumentType` fournissent des informations sur les notations et les entités déclarées par un document (incluant les données externes si l'analyseur les utilise et peut les fournir). Le `DocumentType` d'un `Document` est accessible via l'attribut `doctype`. Si le document ne déclare pas de `DOCTYPE`, l'attribut `doctype` vaudra `None` plutôt qu'une instance de cette interface.

`DocumentType` est une spécialisation de `Node` et ajoute les attributs suivants :

`DocumentType.publicId`

L'identifiant public pour un sous ensemble de la définition type de document (*DTD*). Cela sera une chaîne ou `None`.

`DocumentType.systemId`

L'identifiant système pour un sous ensemble du document de définition type (*DTD*). Cela sera une URI sous la forme d'une chaîne ou `None`.

`DocumentType.internalSubset`

Un chaîne donnant le sous ensemble complet du document. Ceci n'inclut pas les chevrons qui englobe le sous ensemble. Si le document n'a pas de sous ensemble, cela devrait être `None`.

`DocumentType.name`

Le nom de l'élément racine donné dans la déclaration `DOCTYPE` si présente.

`DocumentType.entities`

Ceci est un `NamedNodeMap` donnant les définitions des entités externes. Pour les entités définies plusieurs fois seule la première définition est fournie (les suivantes sont ignorées comme requis par la recommandation XML). Ceci peut retourner `None` si l'information n'est pas fournie à l'analyseur ou si aucune entités n'est définis.

`DocumentType.notations`

Ceci est un `NamedNodeMap` donnant la définition des notations. Pour les notations définies plus d'une fois, seule la première est fournie (les suivantes sont ignorées comme requis par la recommandation XML). Ceci peut retourner `None` si l'information n'est pas fournie à l'analyseur ou si aucune entités n'est définis.

Objets Document

Un `Document` représente un document XML en son entier, incluant les éléments qui le constitue, les attributs, les *processing instructions*, commentaires, etc. Rappelez vous qu'il hérite des propriété de `Node`.

`Document.documentElement`

Le seul et unique élément racine du document.

`Document.createElement (tagName)`

Créé et renvoi un nouveau nœud élément. Ce n'est pas inséré dans le document quand il est créé. Vous avez besoin de l'insérer explicitement avec l'une des autres méthodes comme `insertBefore()` ou `appendChild()`.

`Document.createElementNS (namespaceURI, tagName)`

Créé et renvoi un nouvel élément avec un *namespace*. Le *tagName* peut avoir un préfixe. L'élément ne sera pas insérer dans le document quand il est créé. Vous avez besoin de l'insérer explicitement avec l'une des autres méthodes comme `insertBefore()` ou `appendChild()`.

`Document.createTextNode (data)`

Créé et renvoi un nœud texte contenant les *data* passées en paramètre. Comme pour les autres méthodes de création, la méthode n'insère pas le nœud dans l'arbre.

`Document.createComment (data)`

Créé et renvoi un nœud commentaire contenant les *data* passé en commentaire. Comme pour les autres méthodes de création, la méthode n'insère pas le nœud dans l'arbre.

`Document.createProcessingInstruction(target, data)`

Créé et retourne un nœud *processing instruction* contenant les *target* et *data* passés en paramètres. Comme pour les autres méthodes de création, la méthode n'insère pas le nœud dans l'arbre.

`Document.createAttribute(name)`

Créé et renvoi un nœud attribut. Cette méthode n'associe le nœud attribut aucun nœud en particulier. Vous devez utiliser la méthode `setAttributeNode()` sur un objet `Element` approprié pour utiliser une instance d'attribut nouvellement créé.

`Document.createAttributeNS(namespaceURI, qualifiedName)`

Créé et renvoi un nœud attribut avec un *namespace*. Le *tagName* peut avoir un préfixe. Cette méthode n'associe le nœud attribut à aucun nœud en particulier. Vous devez utiliser la méthode `setAttributeNode()` sur un objet `Element` approprié pour utiliser une instance d'attribut nouvellement créé.

`Document.getElementsByTagName(tagName)`

Cherche tout les descendants (fils directs, fils de fils, etc.) avec un nom de balise particulier.

`Document.getElementsByTagNameNS(namespaceURI, localName)`

Cherche tous les descendants (fils directs, fils de fils, etc.) avec un *namespace URI* particulier et un *localName*. Le *localName* fait parti du *namespace* après le préfixe.

Objets Elements

`Element` est une sous classe de `Node` et donc hérite de tout les éléments de cette classe.

`Element.tagName`

Le nom de l'élément type. Dans un document utilisant des *namespace*, il pourrait y avoir des : dedans. La valeur est une chaîne.

`Element.getElementsByTagName(tagName)`

Identique à la méthode équivalente de la classe `Document`.

`Element.getElementsByTagNameNS(namespaceURI, localName)`

Identique à la méthode équivalente de la classe `Document`.

`Element.hasAttribute(name)`

Renvoi vrai si l'élément a un attribut nommé *name*.

`Element.hasAttributeNS(namespaceURI, localName)`

Renvoi vrai si l'élément a un nommé par *namespaceURI* et *localName*.

`Element.getAttribute(name)`

Retourne la valeur de l'attribut nommé par *name* comme une chaîne. Si un tel attribue n'existe pas, une chaîne vide est retournée comme si l'attribut n'avait aucune valeur.

`Element.getAttributeNode(attrname)`

Retourne le nœud `Attr` pour l'attribut nommé par *attrname*.

`Element.getAttributeNS(namespaceURI, localName)`

Renvoi la valeur de l'attribut nommé par *namespaceURI* et *localName* comme une chaîne. Si un tel attribue n'existe pas, une chaîne vide est retournée comme si l'attribut n'avait aucune valeur.

`Element.getAttributeNodeNS(namespaceURI, localName)`

Renvoi la valeur de l'attribue comme un nœud étant donné *namespaceURI* et *localName*.

`Element.removeAttribute(name)`

Retire un attribut nommé *name*. S'il n'y a aucun attribut correspondant une `NotFoundError` est levée.

`Element.removeAttributeNode(oldAttr)`

Supprime et renvoi *oldAttr* de la liste des attributs si présent. Si *oldAttr* n'est pas présent, `NotFoundError` est levée.

`Element.removeAttributeNS(namespaceURI, localName)`

Retire un attribut selon son nom. Notez qu'il utilise un *localName* et non un *qname*. Aucune exception n'est levée s'il n'y a pas d'attribut correspondant.

`Element.setAttribute(name, value)`

Assigne la valeur à un attribut pour la chaîne.

`Element.setAttributeNode(newAttr)`

Ajoute un nouveau nœud attribut à l'élément, remplaçant un attribut existant si nécessaire si *name* correspond à un attribut. Si l'attribut en remplace un précédent, l'ancien attribut sera retourné. Si *newAttr* est déjà utilisé, *InuseAttributeErr* sera levée.

`Element.setAttributeNodeNS(newAttr)`

Ajoute un nouveau nœud attribut, remplaçant un attribut existant si *namespaceURI* et *localName* correspondent à un attribut. S'il y a remplacement, l'ancien nœud sera renvoyé. Si *newAttr* est déjà utilisé, *InuseAttributeErr* sera levée.

`Element.setAttributeNS(namespaceURI, qname, value)`

Assigne la valeur d'un attribut depuis une chaîne étant donnée un *namespaceURI* et un *qname*. Notez que *qname* est le nom de l'attribut en entier. Ceci est différent d'au dessus.

Objets Attr

`Attr` hérite `Node` et donc hérite de tout ces attributs.

`Attr.name`

Le nom de l'attribut. Dans un document utilisant des *namespaces*, il pourra inclure un `:`.

`Attr.localName`

La partie du nom suivant le `:` s'il y en a un ou le nom entier sinon. Ceci est un attribut en lecture seule.

`Attr.prefix`

La partie du nom précédent le `:` s'il y en a un ou une chaîne vide.

`Attr.value`

La valeur texte de l'attribut. C'est un synonyme de l'attribut `nodeValue`.

Objets NameNodeMap

`NamedNodeMap` n'hérite pas de `Node`.

`NamedNodeMap.length`

La longueur de la liste d'attributs.

`NamedNodeMap.item(index)`

Renvoie un attribut à un index particulier. L'ordre des attribut est arbitraire mais sera constant durant toute la vie du DOM. Chacun des `item` sera un nœud attribut. Obtenez sa valeur avec `value` de l'attribut.

Il y existe également des méthodes expérimentales qui donne à cette classe un comportement plus *mappable*. Vous pouvez les utiliser ou utiliser la famille de méthode standardisés `getAttribute*()` des objets `Element`.

Objets Comment

`Comment` représente un commentaire dans le document XML. C'est une sous classe `Node` mais n'a aucune nœuds fils.

`Comment.data`

Le contenu du commentaire comme une chaîne. L'attribut contient tous les caractères entre `<!--` et `-->` mais ne les inclues pas.

Objets Text et CDATASection

L'interface `Text` représente le texte dans un document XML. Si l'analyseur et l'implémentation DOM supporte les extensions XML du DOM, les portion de texte encapsulées dans des section marquées `CDATA` seront stockées dans des objets `CDATASection`. Ces deux interfaces sont identiques mais fournissent des valeurs différentes pour l'attribut `nodeType`.

Ces interfaces étendent l'interface `Node`. Elles ne peuvent pas avoir de nœuds fils.

`Text.data`

Le contenu du nœud texte comme une chaîne.

Note : L'utilisation d'un nœud `CDATASection` n'indique pas que le nœud représente une section complète marquée `CDATA`, seulement que le contenu du nœud est le contenu d'une section `CDATA`. Une seule section `CDATA` peut représenter plus d'un nœud dans l'arbre du document. Il n'y a aucun moyen de déterminer si deux nœuds `CDATASection` adjacents représentent différentes sections `CDATA`.

Objets ProcessingInstruction

Représente une *processing instruction* dans un document XML. Hérite de l'interface `Node` et ne peut avoir aucun nœud fils.

`ProcessingInstruction.target`

Le contenu de la *processing instruction* jusqu'au premier caractère blanc. Cet attribut est en lecture seule.

`ProcessingInstruction.data`

Le contenu de la *processing instruction* après le premier caractère blanc.

Exceptions

La recommandation *DOM Level 2* définit une seule exception `DOMException` et un nombre de constantes qui permettent aux applications à déterminer quelle type d'erreur s'est produit. Les instances de `DOMException` ont un attribut `code` qui fourni une valeur approprié pour une exception spécifique.

L'interface DOM de Python fournit des constant mais également étends un ensemble d'exception pour qu'il existe une exception spécifique pour chaque code d'exception défini par le DOM. L'implémentation doit lever l'exception spécifique appropriée. Chacune ayant la valeur appropriée pour l'attribut `code`.

exception `xml.dom.DOMException`

Exception de base utilisée pour toutes les exceptions spécifiques du DOM. Cette classe ne peut pas être instanciée directement.

exception `xml.dom.DomstringSizeErr`

Levée quand un intervalle spécifique de texte ne rentre pas dans une chaîne. Cette exception n'est pas réputée être utilisée par les implémentations DOM de Python mais elle peut être levée par des implémentations de DOM qui ne sont pas écrites en Python.

exception `xml.dom.HierarchyRequestErr`

Levée quand l'insertion d'un nœud est tentée dans un type de nœud incompatible.

exception `xml.dom.IndexSizeErr`

Levée quand un index ou la taille d'un paramètre d'une méthode est négatif ou excède les valeurs autorisées.

exception `xml.dom.InuseAttributeErr`

Levée quand l'insertion d'un nœud `Attr` est tenté alors que ce nœud est déjà présent ailleurs dans le document.

exception `xml.dom.InvalidAccessErr`

Levée si un paramètre ou une opération n'est pas supporté par l'objet sous-jacent.

exception `xml.dom.InvalidCharacterErr`

Cette exception est levée quand un paramètre chaîne contient un caractère qui n'est pas autorisé dans le contexte utilisé par la recommandation XML 1.0. Par exemple, lors la tentative de création d'un nœud `Element` avec un espace dans le nom de l'élément.

exception `xml.dom.InvalidModificationErr`

Levée lors de la tentative de modifier le type de nœud.

exception `xml.dom.InvalidStateErr`

Levée quand une tentative est faite d'utiliser un objet non défini ou qui ne sont plus utilisables.

exception `xml.dom.NamespaceErr`

Si une tentative est faite de changer un objet d'une manière qui n'est pas autorisée selon la recommandation [Namespaces in XML](#), cette exception est levée.

exception `xml.dom.NotFoundErr`

Exception quand un nœud n'existe pas dans le contexte référencé. Par exemple, `NamedNodeMap.removeNamedItem()` lèvera cette exception si le nœud passé n'appartient pas à la séquence.

exception `xml.dom.NotSupportedErr`

Levée si l'implémentation ne supporte pas le type d'objet requis ou l'opération.

exception `xml.dom.NoDataAllowedErr`

Levée si la donnée spécifiée pour un nœud n'est pas supportée.

exception `xml.dom.NoModificationAllowedErr`

Levée lors de la tentative de modification sur objet où les modifications ne sont pas autorisées (tels que les nœuds en lecture seule).

exception `xml.dom.SyntaxErr`

Levée quand une chaîne invalide ou illégale est spécifiée.

exception `xml.dom.WrongDocumentErr`

Levée quand un nœud est inséré dans un document différent de celui auquel il appartient et que l'implémentation ne supporte pas la migration d'un document à un autre.

Les codes d'exceptions définis par la recommandation DOM avec leurs correspondances décrites si dessous selon ce tableau :

Constante	Exception
DOMSTRING_SIZE_ERR	<i>DomstringSizeErr</i>
HIERARCHY_REQUEST_ERR	<i>HierarchyRequestErr</i>
INDEX_SIZE_ERR	<i>IndexSizeErr</i>
INUSE_ATTRIBUTE_ERR	<i>InuseAttributeErr</i>
INVALID_ACCESS_ERR	<i>InvalidAccessErr</i>
INVALID_CHARACTER_ERR	<i>InvalidCharacterErr</i>
INVALID_MODIFICATION_ERR	<i>InvalidModificationErr</i>
INVALID_STATE_ERR	<i>InvalidStateErr</i>
NAMESPACE_ERR	<i>NamespaceErr</i>
NOT_FOUND_ERR	<i>NotFoundErr</i>
NOT_SUPPORTED_ERR	<i>NotSupportedErr</i>
NO_DATA_ALLOWED_ERR	<i>NoDataAllowedErr</i>
NO_MODIFICATION_ALLOWED_ERR	<i>NoModificationAllowedErr</i>
SYNTAX_ERR	<i>SyntaxErr</i>
WRONG_DOCUMENT_ERR	<i>WrongDocumentErr</i>

20.6.3 Conformité

Cette section décrit la conformité des pré requis et des relations entre l'API DOM de Python, les recommandations *W3C* DOM et les correspondances *OMG IDL* pour Python.

Correspondance des types

Les types IDL utilisés dans la spécification DOM correspondent aux types Python selon le tableau suivant.

Type IDL	Type Python
boolean	bool ou int
int	int
long int	int
unsigned int	int
DOMString	str or bytes
null	None

Méthodes d'accès

Les correspondance de *OMG IDL* vers Python définissent des fonction d'accès pour les déclarations attribut d'IDL à la manière dont Java le fait. Correspondance des déclarations IDL

```
readonly attribute string someValue;
attribute string anotherValue;
```

Donne trois fonctions d'accès : une méthode *get* pour *someValue* (`_get_someValue()`) et des méthodes *get* et *set* pour *anotherValue* (`_get_anotherValue()` et `_set_anotherValue()`). Le *mapping*, en particulier, ne requiert pas que les attributs IDL soient accessible comme des attributs Python normaux : `object.someValue` n'est pas requis de fonctionner et peut lever une *AttributeError*.

Cependant, l'API DOM de Python impose que les accès par attributs classiques fonctionnent. Par conséquent, les substituts générés par le compilateur IDL de Python ne fonctionneront probablement pas, et des objets façade pourraient être nécessaires côté client si les objets DOM sont manipulés via CORBA. Bien qu'utiliser un client DOM CORBA nécessite

une bonne réflexion, les développeurs habitués et expérimentés à l'utilisation de CORBA ne considèrent pas que c'est un problème. Les attributs déclarés `readonly` pourraient ne pas voir leur accès en écriture restreint dans toutes les implémentations du DOM.

Dans l'API DOM de Python, les fonctions d'accès ne sont pas requises. Si elles sont fournies, elles doivent prendre la forme définie par le *mapping* de Python IDL, mais ces méthodes sont considérées inutiles car les attributs sont directement accessibles depuis Python. Les fonctions d'accès « Set » ne devraient jamais être fournies pour les attributs `readonly` (en lecture seule).

Les définitions IDL n'embarquent pas entièrement les pré-requis de l'API de DOM API telle que la notion de objets ou que la valeur de retour de `getElementsByTagName()` est dynamique. L'API DOM de Python ne requiert pas des implémentations d'avoir de tel pré-requis.

20.7 `xml.dom.minidom` — Minimal DOM implementation

Source code : `Lib/xml/dom/minidom.py`

`xml.dom.minidom` is a minimal implementation of the Document Object Model interface, with an API similar to that in other languages. It is intended to be simpler than the full DOM and also significantly smaller. Users who are not already proficient with the DOM should consider using the `xml.etree.ElementTree` module for their XML processing instead.

Avertissement : The `xml.dom.minidom` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [Vulnérabilités XML](#).

DOM applications typically start by parsing some XML into a DOM. With `xml.dom.minidom`, this is done through the parse functions :

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

The `parse()` function can take either a filename or an open file object.

`xml.dom.minidom.parse(filename_or_file, parser=None, bufsize=None)`

Return a `Document` from the given input. `filename_or_file` may be either a file name, or a file-like object. `parser`, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the `parseString()` function instead :

`xml.dom.minidom.parseString(string, parser=None)`

Return a `Document` that represents the `string`. This method creates an `io.StringIO` object for the string and passes that on to `parse()`.

Both functions return a `Document` object representing the content of the document.

What the `parse()` and `parseString()` functions do is connect an XML parser with a « DOM builder » that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps

misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it's simply that these functions do not provide a parser implementation themselves.

You can also create a `Document` by calling a method on a « DOM Implementation » object. You can get this object either by calling the `getDOMImplementation()` function in the `xml.dom` package or the `xml.dom.minidom` module. Once you have a `Document`, you can add child nodes to it to populate the DOM :

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the `documentElement` property. It gives you the main element in the XML document : the one that holds all others. Here is an example program :

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM tree, you may optionally call the `unlink()` method to encourage early cleanup of the now-unneeded objects. `unlink()` is an `xml.dom.minidom`-specific extension to the DOM API that renders the node and its descendants are essentially useless. Otherwise, Python's garbage collector will eventually take care of the objects in the tree.

Voir aussi :

Spécification Level 1 Document Object Model (DOM) La recommandation du W3C pour le DOM supporté par `xml.dom.minidom`.

20.7.1 DOM Objects

The definition of the DOM API for Python is given as part of the `xml.dom` module documentation. This section lists the differences between the API and `xml.dom.minidom`.

Node.**unlink()**

Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the `Document` object, but may be called on child nodes to discard children of that node.

You can avoid calling this method explicitly by using the `with` statement. The following code will automatically unlink `dom` when the `with` block is exited :

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # Work with dom.
```

Node.**writexml** (*writer*, *indent=""*, *addindent=""*, *newl=""*)

Write XML to the writer object. The writer should have a `write()` method which matches that of the file object interface. The *indent* parameter is the indentation of the current node. The *addindent* parameter is the incremental indentation to use for subnodes of the current one. The *newl* parameter specifies the string to use to terminate newlines.

For the `Document` node, an additional keyword argument *encoding* can be used to specify the encoding field of the XML header.

Node `.toxml` (*encoding=None*)

Return a string or byte string containing the XML represented by the DOM node.

With an explicit *encoding*¹ argument, the result is a byte string in the specified encoding. With no *encoding* argument, the result is a Unicode string, and the XML declaration in the resulting string does not specify an encoding. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

Node `.toprettyxml` (*indent="\t", newl="\n", encoding=None*)

Return a pretty-printed version of the document. *indent* specifies the indentation string and defaults to a tabulator; *newl* specifies the string emitted at the end of each line and defaults to `\n`.

The *encoding* argument behaves like the corresponding argument of `toxml()`.

20.7.2 DOM Example

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
```

(suite sur la page suivante)

1. The encoding name included in the XML output should conform to the appropriate standards. For example, « UTF-8 » is valid, but « UTF8 » is not valid in an XML document's declaration, even though Python accepts it as an encoding name. See <https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl> and <https://www.iana.org/assignments/character-sets/character-sets.xhtml>.

(suite de la page précédente)

```

print("</html>")

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print("<title>%s</title>" % getText(title.childNodes))

def handleSlideTitle(title):
    print("<h2>%s</h2>" % getText(title.childNodes))

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print("<li>%s</li>" % getText(point.childNodes))

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print("<p>%s</p>" % getText(title.childNodes))

handleSlideshow(dom)

```

20.7.3 minidom and the DOM standard

The `xml.dom.minidom` module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply :

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves ; they should use the creator functions available on the `Document` object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.
- Operations are used as methods. Since the DOM uses only `in` parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. `void` operations return `None`.
- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute `foo` can also be accessed through accessor methods `_get_foo()` and `_set_foo()`. `readonly` attributes must not be changed ; this is not enforced at runtime.
- The types `short`, `int`, `unsigned int`, `unsigned long`, `long`, and `boolean` all map to Python integer objects.
- The type `DOMString` maps to Python strings. `xml.dom.minidom` supports either bytes or strings, but will normally produce strings. Values of type `DOMString` may also be `None` where allowed to have the IDL null value by the DOM specification from the W3C.
- `const` declarations map to variables in their respective scope (e.g. `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`) ; they must not be changed.

- `DOMException` is currently not supported in `xml.dom.minidom`. Instead, `xml.dom.minidom` uses standard Python exceptions such as `TypeError` and `AttributeError`.
- `NodeList` objects are implemented using Python's built-in list type. These objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more « Pythonic » than the interface defined in the W3C recommendations.

The following interfaces have no implementation in `xml.dom.minidom`:

- `DOMTimeStamp`
- `DocumentType`
- `DOMImplementation`
- `CharacterData`
- `CDATASection`
- `Notation`
- `Entity`
- `EntityReference`
- `DocumentFragment`

Most of these reflect information in the XML document that is not of general utility to most DOM users.

Notes

20.8 `xml.dom.pulldom` — Support for building partial DOM trees

Source code : <Lib/xml/dom/pulldom.py>

The `xml.dom.pulldom` module provides a « pull parser » which can also be asked to produce DOM-accessible fragments of the document where necessary. The basic concept involves pulling « events » from a stream of incoming XML and processing them. In contrast to SAX which also employs an event-driven processing model together with callbacks, the user of a pull parser is responsible for explicitly pulling events from the stream, looping over those events until either processing is finished or an error condition occurs.

Avertissement : The `xml.dom.pulldom` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [Vulnérabilités XML](#).

Modifié dans la version 3.6.7 : The SAX parser no longer processes general external entities by default to increase security by default. To enable processing of external entities, pass a custom parser instance in :

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

Exemple :

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
```

(suite sur la page suivante)

(suite de la page précédente)

```

if int(node.getAttribute('price')) > 50:
    doc.expandNode(node)
print(node.toxml())

```

`event` is a constant and can be one of :

- `START_ELEMENT`
- `END_ELEMENT`
- `COMMENT`
- `START_DOCUMENT`
- `END_DOCUMENT`
- `CHARACTERS`
- `PROCESSING_INSTRUCTION`
- `IGNORABLE_WHITESPACE`

`node` is an object of type `xml.dom.minidom.Document`, `xml.dom.minidom.Element` or `xml.dom.minidom.Text`.

Since the document is treated as a « flat » stream of events, the document « tree » is implicitly traversed and the desired elements are found regardless of their depth in the tree. In other words, one does not need to consider hierarchical issues such as recursive searching of the document nodes, although if the context of elements were important, one would either need to maintain some context-related state (i.e. remembering where one is in the document at any given point) or to make use of the `DOMEventStream.expandNode()` method and switch to DOM-related processing.

class `xml.dom.pulldom.PullDom` (*documentFactory=None*)
 Subclass of `xml.sax.handler.ContentHandler`.

class `xml.dom.pulldom.SAX2DOM` (*documentFactory=None*)
 Subclass of `xml.sax.handler.ContentHandler`.

`xml.dom.pulldom.parse` (*stream_or_string*, *parser=None*, *bufsize=None*)
 Return a `DOMEventStream` from the given input. *stream_or_string* may be either a file name, or a file-like object. *parser*, if given, must be an `XMLReader` object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the `parseString()` function instead :

`xml.dom.pulldom.parseString` (*string*, *parser=None*)
 Return a `DOMEventStream` that represents the (Unicode) *string*.

`xml.dom.pulldom.default_bufsize`
 Default value for the *bufsize* parameter to `parse()`.
 The value of this variable can be changed before calling `parse()` and the new value will take effect.

20.8.1 DOMEventStream Objects

class `xml.dom.pulldom.DOMEventStream` (*stream*, *parser*, *bufsize*)

getEvent ()
 Return a tuple containing *event* and the current *node* as `xml.dom.minidom.Document` if *event* equals `START_DOCUMENT`, `xml.dom.minidom.Element` if *event* equals `START_ELEMENT` or `END_ELEMENT` or `xml.dom.minidom.Text` if *event* equals `CHARACTERS`. The current node does not contain information about its children, unless `expandNode()` is called.

expandNode (*node*)
 Expands all children of *node* into *node*. Example :

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<p>Some text
        ↪<div>and more</div></p>'
        print(node.toxml())
```

`reset()`

20.9 `xml.sax` — Prise en charge des analyseurs SAX2

Code source : `Lib/xml/sax/__init__.py`

Le paquet `xml.sax` fournit des modules qui implémentent l'interface *Simple API for XML (SAX)* pour Python. Le paquet en lui-même fournit les exceptions SAX et les fonctions les plus utiles qui seront le plus utilisées par les utilisateurs de SAX API.

Avertissement : Le module `xml.sax` n'est pas sécurisé contre les données construites de façon malveillante. Si vous avez besoin d'analyser des données non sécurisées ou non authentifiées, voir [Vulnérabilités XML](#).

Modifié dans la version 3.6.7 : The SAX parser no longer processes general external entities by default to increase security. Before, the parser created network connections to fetch remote files or loaded local files from the file system for DTD and entities. The feature can be enabled again with method `setFeature()` on the parser object and argument `feature_external_ges`.

Les fonctions les plus utiles sont :

`xml.sax.make_parser(parser_list=[])`

Create and return a SAX `XMLReader` object. The first parser found will be used. If `parser_list` is provided, it must be a list of strings which name modules that have a function named `create_parser()`. Modules listed in `parser_list` will be used before modules in the default list of parsers.

`xml.sax.parse(filename_or_stream, handler, error_handler=handler.ErrorHandler())`

Crée un analyseur SAX et l'utilise pour analyser un document. Le document transmis comme `filename_or_stream`, peut être un nom de fichier ou un objet fichier. Le paramètre `handler` doit être une instance SAX `ContentHandler`. Si un `error_handler` est donné, il doit être un SAX `ErrorHandler`; si omis, `SAXParseException` sera levé sur toutes les erreurs. Il n'y a pas de valeur de retour, tout le travail doit être fait par le `handler` transmis.

`xml.sax.parseString(string, handler, error_handler=handler.ErrorHandler())`

Similaire à `parse()`, mais qui analyse à partir d'un espace mémoire `string` reçu en tant que paramètre. `string` doit être une instance `str` ou un objet *bytes-like object*.

Modifié dans la version 3.5 : Ajout du support des instances `str`.

Une application SAX typique utilise trois types d'objets : les *readers*, les *handlers* et les sources d'entrée. « Reader » dans ce contexte est un autre terme pour le analyseur, c'est-à-dire un morceau de code qui lit les octets ou les caractères de la source d'entrée et qui produit une séquence d'événements. Les événements sont ensuite distribués aux objets du *handler*,

c'est-à-dire que le lecteur appelle une méthode sur le *handler*. L'application doit donc obtenir un objet *reader*, créer ou ouvrir les sources d'entrée, créer les *handlers* et connecter ces objets tous ensemble. La dernière étape de la préparation, le *reader* est appelé à analyser l'entrée. Pendant l'analyse, les méthodes sur les objets du *handler* sont appelées en fonction d'événements structurels et syntaxiques à partir des données d'entrée.

Pour ces objets, seules les interfaces sont pertinentes; elles ne sont pas normalement instanciées par l'application elle-même. Puisque Python n'a pas de notion explicite d'interface, elles sont formellement introduites en tant que classes, mais les applications peuvent utiliser des implémentations qui n'héritent pas des classes fournies. Les interfaces *InputSource*, *Locator*, *Attributes*, *AttributesNS*, et *XMLReader* sont définies dans le module *xml.sax.xmlreader*. Les interfaces du *handler* sont définies dans *xml.sax.handler*. Pour plus de commodité, *xml.sax.xmlreader.InputSource* (qui est souvent instancié directement) et les classes du *handler* sont également disponibles à partir de *xml.sax*. Ces interfaces sont décrites ci-dessous.

En plus de ces classes, *xml.sax* fournit les classes d'exceptions suivantes.

exception *xml.sax.SAXException* (*msg*, *exception=None*)

Encapsule une erreur ou un avertissement XML. Cette classe peut contenir une erreur de base ou une information d'avertissement soit du analyseur XML ou de l'application : elle peut être sous-classée pour fournir des fonctionnalités supplémentaires ou pour ajouter une localisation. Noter que même si les *handlers* définis dans l'interface *ErrorHandler* reçoivent des instances de cette exception, ce n'est pas nécessaire de lever l'exception — il est également utile en tant que conteneur pour l'information.

Quand instancié, *msg* devrait être une description lisible par l'homme de l'erreur. Le paramètre optionnel *exception*, s'il est donné, devrait être *None* ou une exception qui a été interceptée par le code d'analyse et qui est transmise comme information.

Ceci est la classe de base pour les autres classes d'exception SAX.

exception *xml.sax.SAXParseException* (*msg*, *exception*, *locator*)

Sous-classe de *SAXException* élevée sur les erreurs d'analyse. Les instances de cette classe sont passées aux méthodes de l'interface SAX *ErrorHandler* pour fournir des informations sur l'erreur d'analyse. Cette classe supporte aussi l'interface SAX *Locator* comme l'interface *SAXException*.

exception *xml.sax.SAXNotRecognizedException* (*msg*, *exception=None*)

Sous-classe de *SAXException* levée quand un SAX *XMLReader* est confronté à une caractéristique ou à une propriété non reconnue. Les applications et les extensions SAX peuvent utiliser cette classe à des fins similaires.

exception *xml.sax.SAXNotSupportedException* (*msg*, *exception=None*)

Sous-classe de *SAXException* levée quand un SAX *XMLReader* est demandé pour activer une fonctionnalité qui n'est pas supportée, ou pour définir une propriété à une valeur que l'implémentation ne prend pas en charge. Les applications et les extensions SAX peuvent utiliser cette classe à des fins similaires.

Voir aussi :

SAX : L'API simple pour XML Ce site est le point focal pour la définition de l'API SAX. Il offre une implémentation Java et une documentation en ligne. Des liens pour l'implémentation et des informations historiques sont également disponibles.

Module *xml.sax.handler* Définitions des interfaces pour les objets fournis par l'application.

Module *xml.sax.saxutils* Fonctions pratiques pour une utilisation dans les applications SAX.

Module *xml.sax.xmlreader* Définitions des interfaces pour les objets fournis par l'analyseur.

20.9.1 Les objets `SAXException`

La classe d'exception `SAXException` supporte les méthodes suivantes :

`SAXException.getMessage()`

Renvoyer un message lisible par l'homme décrivant la condition d'erreur.

`SAXException.getException()`

Renvoie un objet d'exception encapsulé, ou "None".

20.10 `xml.sax.handler` — Base classes for SAX handlers

Source code : <Lib/xml/sax/handler.py>

The SAX API defines four kinds of handlers : content handlers, DTD handlers, error handlers, and entity resolvers. Applications normally only need to implement those interfaces whose events they are interested in ; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module `xml.sax.handler`, so that all methods get default implementations.

class `xml.sax.handler.ContentHandler`

This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class `xml.sax.handler.DTDHandler`

Handle DTD events.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class `xml.sax.handler.EntityResolver`

Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

class `xml.sax.handler.ErrorHandler`

Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

In addition to these classes, `xml.sax.handler` provides symbolic constants for the feature and property names.

`xml.sax.handler.feature_namespaces`

value : "http://xml.org/sax/features/namespaces"

true : Perform Namespace processing.

false : Optionally do not perform Namespace processing (implies namespace-prefixes ; default).

access : (parsing) read-only ; (not parsing) read/write

`xml.sax.handler.feature_namespace_prefixes`

value : "http://xml.org/sax/features/namespace-prefixes"

true : Report the original prefixed names and attributes used for Namespace declarations.

false : Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).

access : (parsing) read-only ; (not parsing) read/write

`xml.sax.handler.feature_string_interning`

value : "http://xml.org/sax/features/string-interning"
 true : All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
 false : Names are not necessarily interned, although they may be (default).
 access : (parsing) read-only ; (not parsing) read/write

`xml.sax.handler.feature_validation`

value : "http://xml.org/sax/features/validation"
 true : Report all validation errors (implies external-general-entities and external-parameter-entities).
 false : Do not report validation errors.
 access : (parsing) read-only ; (not parsing) read/write

`xml.sax.handler.feature_external_ges`

value : "http://xml.org/sax/features/external-general-entities"
 true : Include all external general (text) entities.
 false : Do not include external general entities.
 access : (parsing) read-only ; (not parsing) read/write

`xml.sax.handler.feature_external_pes`

value : "http://xml.org/sax/features/external-parameter-entities"
 true : Include all external parameter entities, including the external DTD subset.
 false : Do not include any external parameter entities, even the external DTD subset.
 access : (parsing) read-only ; (not parsing) read/write

`xml.sax.handler.all_features`

List of all features.

`xml.sax.handler.property_lexical_handler`

value : "http://xml.org/sax/properties/lexical-handler"
 data type : `xml.sax.sax2lib.LexicalHandler` (not supported in Python 2)
 description : An optional extension handler for lexical events like comments.
 access : read/write

`xml.sax.handler.property_declaration_handler`

value : "http://xml.org/sax/properties/declaration-handler"
 data type : `xml.sax.sax2lib.DeclHandler` (not supported in Python 2)
 description : An optional extension handler for DTD-related events other than notations and unparsed entities.
 access : read/write

`xml.sax.handler.property_dom_node`

value : "http://xml.org/sax/properties/dom-node"
 data type : `org.w3c.dom.Node` (not supported in Python 2)
 description : When parsing, the current DOM node being visited if this is a DOM iterator ; when not parsing, the root DOM node for iteration.
 access : (parsing) read-only ; (not parsing) read/write

`xml.sax.handler.property_xml_string`

`value` : "http://xml.org/sax/properties/xml-string"
`data type` : String
`description` : The literal string of characters that was the source for the current event.
`access` : read-only

`xml.sax.handler.all_properties`
List of all known property names.

20.10.1 ContentHandler Objects

Users are expected to subclass `ContentHandler` to support their application. The following methods are called by the parser on the appropriate events in the input document :

`ContentHandler.setDocumentLocator(locator)`

Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator : if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the `DocumentHandler` interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

`ContentHandler.startDocument()`

Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in `DTDHandler` (except for `setDocumentLocator()`).

`ContentHandler.endDocument()`

Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

`ContentHandler.startPrefixMapping(prefix, uri)`

Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing : the SAX XML reader will automatically replace prefixes for element and attribute names when the `feature_namespaces` feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically ; the `startPrefixMapping()` and `endPrefixMapping()` events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that `startPrefixMapping()` and `endPrefixMapping()` events are not guaranteed to be properly nested relative to each-other : all `startPrefixMapping()` events will occur before the corresponding `startElement()` event, and all `endPrefixMapping()` events will occur after the corresponding `endElement()` event, but their order is not guaranteed.

`ContentHandler.endPrefixMapping(prefix)`

End the scope of a prefix-URI mapping.

See `startPrefixMapping()` for details. This event will always occur after the corresponding `endElement()` event, but the order of `endPrefixMapping()` events is not otherwise guaranteed.

`ContentHandler.startElement` (*name*, *attrs*)

Signals the start of an element in non-namespace mode.

The *name* parameter contains the raw XML 1.0 name of the element type as a string and the *attrs* parameter holds an object of the `Attributes` interface (see [The Attributes Interface](#)) containing the attributes of the element. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the `copy()` method of the *attrs* object.

`ContentHandler.endElement` (*name*)

Signals the end of an element in non-namespace mode.

The *name* parameter contains the name of the element type, just as with the `startElement()` event.

`ContentHandler.startElementNS` (*name*, *qname*, *attrs*)

Signals the start of an element in namespace mode.

The *name* parameter contains the name of the element type as a (`uri`, `localname`) tuple, the *qname* parameter contains the raw XML 1.0 name used in the source document, and the *attrs* parameter holds an instance of the `AttributesNS` interface (see [The AttributesNS Interface](#)) containing the attributes of the element. If no namespace is associated with the element, the *uri* component of *name* will be `None`. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the `copy()` method of the *attrs* object.

Parsers may set the *qname* parameter to `None`, unless the `feature_namespace_prefixes` feature is activated.

`ContentHandler.endElementNS` (*name*, *qname*)

Signals the end of an element in namespace mode.

The *name* parameter contains the name of the element type, just as with the `startElementNS()` method, likewise the *qname* parameter.

`ContentHandler.characters` (*content*)

Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

content may be a string or bytes instance; the `expat` reader module always produces strings.

Note : The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use *content* instead of slicing content with the old *offset* and *length* parameters.

`ContentHandler.ignorableWhitespace` (*whitespace*)

Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10) : non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

`ContentHandler.processingInstruction` (*target*, *data*)

Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found : note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

`ContentHandler.skippedEntity(name)`

Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the `feature_external_ges` and the `feature_external_pes` properties.

20.10.2 DTDHandler Objects

DTDHandler instances provide the following methods :

`DTDHandlernotationDecl(name, publicId, systemId)`

Handle a notation declaration event.

`DTDHandlerunparsedEntityDecl(name, publicId, systemId, ndata)`

Handle an unparsed entity declaration event.

20.10.3 EntityResolver Objects

`EntityResolver.resolveEntity(publicId, systemId)`

Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an `InputSource` to read from. The default implementation returns *systemId*.

20.10.4 ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the *XMLReader*. If you create an object that implements this interface, then register the object with your *XMLReader*, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available : warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a `SAXParseException` as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

`ErrorHandler.error(exception)`

Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

`ErrorHandler.fatalError(exception)`

Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

`ErrorHandler.warning(exception)`

Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.

20.11 `xml.sax.saxutils` — Utilitaires SAX

Code source : `Lib/xml/sax/saxutils.py`

Le module `xml.sax.saxutils` contient des classes et fonctions qui sont fréquemment utiles en créant des applications SAX, soit en utilisation directe, soit en classes de base.

`xml.sax.saxutils.escape(data, entities={})`

Échappe '&', '<', et '>' dans une chaîne de caractères de données.

Vous pouvez échapper d'autres chaînes de caractères de données en passant un dictionnaire au paramètre optionnel *entities*. Les clés et valeurs doivent toutes être des chaînes de caractères ; chaque clé sera remplacée par sa valeur correspondante. Les caractères '&', '<' et '>' sont toujours échappés même si *entities* est donné en paramètre.

`xml.sax.saxutils.unescape(data, entities={})`

Parse '&'; '<'; et '>' dans une chaîne de caractères de données.

Vous pouvez dé-échapper d'autres chaînes de caractères de données en passant un dictionnaire au paramètre optionnel *entities*. Les clés et valeurs doivent toutes être des chaînes de caractères ; chaque clé sera remplacée par sa valeur correspondante. Les caractères '&', '<' et '>' sont toujours dé-échappés même si *entities* est donné en paramètre.

`xml.sax.saxutils.quoteattr(data, entities={})`

Similaire à `escape()`, mais prépare aussi *data* pour être utilisé comme une valeur d'attribut. La valeur renvoyée est une version entre guillemets de *data* avec tous les remplacements supplémentaires nécessaires. `quoteattr()` va sélectionner un caractère guillemet basé sur le contenu de *data*, en essayant d'éviter d'encoder tous les caractères guillemets dans la chaîne de caractères. Si les caractères guillemet simple et guillemets sont déjà dans *data*, les caractères guillemets simples seront encodés et *data* sera entouré de guillemets. La chaîne de caractères résultante pourra être utilisée en tant que valeur d'attribut :

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

Cette fonction est utile quand vous générez des valeurs d'attributs pour du HTML ou n'importe quel SGML en utilisant la syntaxe concrète de référence.

class `xml.sax.saxutils.XMLGenerator` (*out=None*, *encoding='iso-8859-1'*,
short_empty_elements=False)

Cette classe implémente l'interface *ContentHandler* en écrivant les événements SAX dans un document XML. En d'autres termes, utiliser un *XMLGenerator* en tant que gestionnaire de contenu reproduira le document original qui était analysé. *out* devrait être un objet de type fichier qui est par défaut `sys.stdout`. *encoding* est l'encodage du flux de sortie qui est par défaut `'iso-8859-1'`. *short_empty_elements* contrôle le formatage des éléments qui ne contiennent rien : si `False` (par défaut), ils sont émis comme une paire de balises (début, fin). Si la valeur est `True`, ils sont émis comme une balise seule auto-fermante.

Nouveau dans la version 3.2 : Le paramètre *short_empty_elements*.

class `xml.sax.saxutils.XMLFilterBase` (*base*)

Cette classe est faite pour être entre *XMLReader* et le gestionnaire des événements de l'application client. Par défaut, elle ne fait rien mais passe les requêtes au lecteur et les événements au gestionnaire sans les modifier, mais des sous-classes peuvent surcharger des méthodes spécifiques pour modifier le flux d'événements ou la configuration des requêtes à leur passage.

`xml.sax.saxutils.prepare_input_source(source, base="")`

Cette fonction prend en entrée une source et une URL de base optionnelle et retourne un objet complètement résolue *InputSource* prêt pour être lu. La source d'entrée peut être donnée comme une chaîne de caractère, un objet type fichier, ou un objet *InputSource* ; Les analyseurs utiliseront cette fonction pour gérer le polymorphisme de l'argument *source* à leur méthode `parse()`.

20.12 `xml.sax.xmlreader` — Interface for XML parsers

Source code : [Lib/xml/sax/xmlreader.py](#)

SAX parsers implement the *XMLReader* interface. They are implemented in a Python module, which must provide a function `create_parser()`. This function is invoked by `xml.sax.make_parser()` with no arguments to create a new parser object.

class `xml.sax.xmlreader.XMLReader`
Base class which can be inherited by SAX parsers.

class `xml.sax.xmlreader.IncrementalParser`
In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still `parse()` won't return until the entire document is processed. So these interfaces should be used if the blocking behaviour of `parse()` is not desirable.

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must *not* be called during parsing, that is, after `parse` has been called and before it returns. By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

class `xml.sax.xmlreader.Locator`
Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return `None`.

class `xml.sax.xmlreader.InputSource` (*system_id=None*)
Encapsulation of the information needed by the *XMLReader* to read entities.
This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.
Applications will create objects of this class for use in the *XMLReader.parse()* method and for returning from `EntityResolver.resolveEntity`.
An *InputSource* belongs to the application, the *XMLReader* is not allowed to modify *InputSource* objects passed to it from the application, although it may make copies and modify those.

class `xml.sax.xmlreader.AttributesImpl` (*attrs*)
This is an implementation of the *Attributes* interface (see section *The Attributes Interface*). This is a dictionary-like object which represents the element attributes in a `startElement()` call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers; *attrs* must be a dictionary-like object containing a mapping from attribute names to attribute values.

class `xml.sax.xmlreader.AttributesNSImpl` (*attrs, qnames*)
Namespace-aware variant of *AttributesImpl*, which will be passed to `startElementNS()`. It is derived from *AttributesImpl*, but understands attribute names as two-tuples of *namespaceURI* and *localname*. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the *AttributesNS* interface (see section *The AttributesNS Interface*).

20.12.1 XMLReader Objects

The *XMLReader* interface supports the following methods :

`XMLReader.parse(source)`

Process an input source, producing SAX events. The *source* object can be a system identifier (a string identifying the input source – typically a file name or a URL), a file-like object, or an *InputSource* object. When *parse()* returns, the input is completely processed, and the parser object can be discarded or reset.

Modifié dans la version 3.5 : Added support of character streams.

`XMLReader.getContentHandler()`

Return the current *ContentHandler*.

`XMLReader.setContentHandler(handler)`

Set the current *ContentHandler*. If no *ContentHandler* is set, content events will be discarded.

`XMLReader.getDTDHandler()`

Return the current *DTDHandler*.

`XMLReader.setDTDHandler(handler)`

Set the current *DTDHandler*. If no *DTDHandler* is set, DTD events will be discarded.

`XMLReader.getEntityResolver()`

Return the current *EntityResolver*.

`XMLReader.setEntityResolver(handler)`

Set the current *EntityResolver*. If no *EntityResolver* is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

`XMLReader.getErrorHandler()`

Return the current *ErrorHandler*.

`XMLReader.setErrorHandler(handler)`

Set the current error handler. If no *ErrorHandler* is set, errors will be raised as exceptions, and warnings will be printed.

`XMLReader.setLocale(locale)`

Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings ; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.

`XMLReader.getFeature(featurename)`

Return the current setting for feature *featurename*. If the feature is not recognized, *SAXNotRecognizedException* is raised. The well-known featurenames are listed in the module *xml.sax.handler*.

`XMLReader.setFeature(featurename, value)`

Set the *featurename* to *value*. If the feature is not recognized, *SAXNotRecognizedException* is raised. If the feature or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

`XMLReader.getProperty(propertyname)`

Return the current setting for property *propertyname*. If the property is not recognized, a *SAXNotRecognizedException* is raised. The well-known propertynames are listed in the module *xml.sax.handler*.

`XMLReader.setProperty(propertyname, value)`

Set the *propertyname* to *value*. If the property is not recognized, *SAXNotRecognizedException* is raised. If the property or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

20.12.2 IncrementalParser Objects

Instances of *IncrementalParser* offer the following additional methods :

`IncrementalParser.feed(data)`

Process a chunk of *data*.

`IncrementalParser.close()`

Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

`IncrementalParser.reset()`

This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.

20.12.3 Locator Objects

Instances of *Locator* provide these methods :

`Locator.getColumnNumber()`

Return the column number where the current event begins.

`Locator.getLineNumber()`

Return the line number where the current event begins.

`Locator.getPublicId()`

Return the public identifier for the current event.

`Locator.getSystemId()`

Return the system identifier for the current event.

20.12.4 InputSource Objects

`InputSource.setPublicId(id)`

Sets the public identifier of this *InputSource*.

`InputSource.getPublicId()`

Returns the public identifier of this *InputSource*.

`InputSource.setSystemId(id)`

Sets the system identifier of this *InputSource*.

`InputSource.getSystemId()`

Returns the system identifier of this *InputSource*.

`InputSource.setEncoding(encoding)`

Sets the character encoding of this *InputSource*.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the *InputSource* is ignored if the *InputSource* also contains a character stream.

`InputSource.getEncoding()`

Get the character encoding of this *InputSource*.

`InputSource.setByteStream(bytefile)`

Set the byte stream (a *binary file*) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.

If the application knows the character encoding of the byte stream, it should set it with the `setEncoding` method.

`InputSource.getByteStream()`

Get the byte stream for this input source.

The `getEncoding` method will return the character encoding for this byte stream, or `None` if unknown.

`InputSource.setCharacterStream(charfile)`

Set the character stream (a *text file*) for this input source.

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

`InputSource.getCharacterStream()`

Get the character stream for this input source.

20.12.5 The `Attributes` Interface

`Attributes` objects implement a portion of the *mapping protocol*, including the methods `copy()`, `get()`, `__contains__()`, `items()`, `keys()`, and `values()`. The following methods are also provided :

`Attributes.getLength()`

Return the number of attributes.

`Attributes.getNames()`

Return the names of the attributes.

`Attributes.getType(name)`

Returns the type of the attribute *name*, which is normally `'CDATA'`.

`Attributes.getValue(name)`

Return the value of attribute *name*.

20.12.6 The `AttributesNS` Interface

This interface is a subtype of the `Attributes` interface (see section *The Attributes Interface*). All methods supported by that interface are also available on `AttributesNS` objects.

The following methods are also available :

`AttributesNS.getValueByQName(name)`

Return the value for a qualified name.

`AttributesNS.getNameByQName(name)`

Return the (namespace, localname) pair for a qualified *name*.

`AttributesNS.getQNameByName(name)`

Return the qualified name for a (namespace, localname) pair.

`AttributesNS.getQNames()`

Return the qualified names of all attributes.

20.13 `xml.parsers.expat` — Fast XML parsing using Expat

Avertissement : The `pyexpat` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [Vulnérabilités XML](#).

The `xml.parsers.expat` module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, `xmlparser`, that represents the current state of an XML parser. After an `xmlparser` object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the `pyexpat` module to provide access to the Expat parser. Direct use of the `pyexpat` module is deprecated.

This module provides one exception and one type object :

exception `xml.parsers.expat.ExpatError`

The exception raised when Expat reports an error. See section [ExpatError Exceptions](#) for more information on interpreting Expat errors.

exception `xml.parsers.expat.error`

Alias for [ExpatError](#).

`xml.parsers.expat.XMLParserType`

The type of the return values from the [ParserCreate\(\)](#) function.

The `xml.parsers.expat` module contains two functions :

`xml.parsers.expat.ErrorString(errno)`

Returns an explanatory string for a given error number *errno*.

`xml.parsers.expat.ParserCreate(encoding=None, namespace_separator=None)`

Creates and returns a new `xmlparser` object. *encoding*, if specified, must be a string naming the encoding used by the XML data. Expat doesn't support as many encodings as Python does, and its repertoire of encodings can't be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If *encoding*¹ is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for *namespace_separator*. The value must be a one-character string; a [ValueError](#) will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers `StartElementHandler` and `EndElementHandler` will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (`chr(0)`) then the namespace URI and the local part will be concatenated without any separator.

For example, if *namespace_separator* is set to a space character (' ') and the following document is parsed :

```
<?xml version="1.0"?>
<root xmlns      = "http://default-namespace.org/"
      xmlns:py   = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

`StartElementHandler` will receive the following strings for each element :

1. The encoding string included in XML output should conform to the appropriate standards. For example, « UTF-8 » is valid, but « UTF8 » is not. See <https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl> and <https://www.iana.org/assignments/character-sets/character-sets.xhtml>.


```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Due to limitations in the Expat library used by `pyexpat`, the `xmlparser` instance returned can only be used to parse a single XML document. Call `ParserCreate` for each document to provide unique parser instances.

Voir aussi :

[The Expat XML Parser](#) Home page of the Expat project.

20.13.1 XMLParser Objects

`xmlparser` objects have the following methods :

`xmlparser.Parse(data[, isfinal])`

Parses the contents of the string *data*, calling the appropriate handler functions to process the parsed data. *isfinal* must be true on the final call to this method; it allows the parsing of a single file in fragments, not the submission of multiple files. *data* can be the empty string at any time.

`xmlparser.ParseFile(file)`

Parse XML data reading from the object *file*. *file* only needs to provide the `read(nbytes)` method, returning the empty string when there's no more data.

`xmlparser.SetBase(base)`

Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application : this value will be passed through as the *base* argument to the `ExternalEntityRefHandler()`, `NotationDeclHandler()`, and `UnparsedEntityDeclHandler()` functions.

`xmlparser.GetBase()`

Returns a string containing the base set by a previous call to `SetBase()`, or `None` if `SetBase()` hasn't been called.

`xmlparser.GetInputContext()`

Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is `None`.

`xmlparser.ExternalEntityParserCreate(context[, encoding])`

Create a « child » parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The *context* parameter should be the string passed to the `ExternalEntityRefHandler()` handler function, described below. The child parser is created with the `ordered_attributes` and `specified_attributes` set to the values of this parser.

`xmlparser.SetParamEntityParsing(flag)`

Control parsing of parameter entities (including the external DTD subset). Possible *flag* values are `XML_PARAM_ENTITY_PARSING_NEVER`, `XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE` and `XML_PARAM_ENTITY_PARSING_ALWAYS`. Return true if setting the flag was successful.

`xmlparser.UseForeignDTD([flag])`

Calling this with a true value for *flag* (the default) will cause Expat to call the `ExternalEntityRefHandler` with `None` for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the `ExternalEntityRefHandler` will still be called, but the `StartDoctypeDeclHandler` and `EndDoctypeDeclHandler` will not be called.

Passing a false value for *flag* will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the `Parse()` or `ParseFile()` methods are called; calling it after either of those have been called causes `ExpatriError` to be raised with the `code` attribute set to `errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]`.

`xmlparser` objects have the following attributes :

`xmlparser.buffer_size`

The size of the buffer used when *buffer_text* is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

`xmlparser.buffer_text`

Setting this to true causes the `xmlparser` object to buffer textual content returned by Expat to avoid multiple calls to the *CharacterDataHandler()* callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time.

`xmlparser.buffer_used`

If *buffer_text* is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when *buffer_text* is false.

`xmlparser.ordered_attributes`

Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented : the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false ; it may be changed at any time.

`xmlparser.specified_attributes`

If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false ; it may be changed at any time.

The following attributes contain values relating to the most recent error encountered by an `xmlparser` object, and will only have correct values once a call to `Parse()` or `ParseFile()` has raised an *xml.parsers.expat.ExpatError* exception.

`xmlparser.ErrorByteIndex`

Byte index at which an error occurred.

`xmlparser.ErrorCode`

Numeric code specifying the problem. This value can be passed to the *ErrorString()* function, or compared to one of the constants defined in the `errors` object.

`xmlparser.ErrorColumnNumber`

Column number at which an error occurred.

`xmlparser.ErrorLineNumber`

Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an `xmlparser` object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

`xmlparser.CurrentByteIndex`

Current byte index in the parser input.

`xmlparser.CurrentColumnNumber`

Current column number in the parser input.

`xmlparser.CurrentLineNumber`

Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an `xmlparser` object *o*, use `o.handlername = func`. *handlername* must be taken from the following list, and *func* must be a callable object accepting the correct

number of arguments. The arguments are all strings, unless otherwise stated.

`xmlparser.XmlDeclHandler` (*version, encoding, standalone*)

Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional « standalone » declaration. *version* and *encoding* will be strings, and *standalone* will be 1 if the document is declared standalone, 0 if it is declared not to be standalone, or -1 if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

`xmlparser.StartDoctypeDeclHandler` (*doctypeName, systemId, publicId, has_internal_subset*)

Called when Expat begins parsing the document type declaration (`<!DOCTYPE . . .`). The *doctypeName* is provided exactly as presented. The *systemId* and *publicId* parameters give the system and public identifiers if specified, or `None` if omitted. *has_internal_subset* will be true if the document contains an internal document declaration subset. This requires Expat version 1.2 or newer.

`xmlparser.EndDoctypeDeclHandler` ()

Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

`xmlparser.ElementDeclHandler` (*name, model*)

Called once for each element type declaration. *name* is the name of the element type, and *model* is a representation of the content model.

`xmlparser.AttnlistDeclHandler` (*elname, attname, type, default, required*)

Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. *elname* is the name of the element to which the declaration applies and *attname* is the name of the attribute declared. The attribute type is a string passed as *type*; the possible values are 'CDATA', 'ID', 'IDREF', ... *default* gives the default value for the attribute used when the attribute is not specified by the document instance, or `None` if there is no default value (#IMPLIED values). If the attribute is required to be given in the document instance, *required* will be true. This requires Expat version 1.95.0 or newer.

`xmlparser.StartElementHandler` (*name, attributes*)

Called for the start of every element. *name* is a string containing the element name, and *attributes* is the element attributes. If *ordered_attributes* is true, this is a list (see *ordered_attributes* for a full description). Otherwise it's a dictionary mapping names to values.

`xmlparser.EndElementHandler` (*name*)

Called for the end of every element.

`xmlparser.ProcessingInstructionHandler` (*target, data*)

Called for every processing instruction.

`xmlparser.CharacterDataHandler` (*data*)

Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the *StartCdataSectionHandler*, *EndCdataSectionHandler*, and *ElementDeclHandler* callbacks to collect the required information.

`xmlparser.UnparsedEntityDeclHandler` (*entityName, base, systemId, publicId, notationName*)

Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use *EntityDeclHandler* instead. (The underlying function in the Expat library has been declared obsolete.)

`xmlparser.EntityDeclHandler` (*entityName, is_parameter_entity, value, base, systemId, publicId, notationName*)

Called for all entity declarations. For parameter and internal entities, *value* will be a string giving the declared contents of the entity; this will be `None` for external entities. The *notationName* parameter will be `None` for parsed entities, and the name of the notation for unparsed entities. *is_parameter_entity* will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

`xmlparser.NotationDeclHandler` (*notationName*, *base*, *systemId*, *publicId*)

Called for notation declarations. *notationName*, *base*, and *systemId*, and *publicId* are strings if given. If the public identifier is omitted, *publicId* will be `None`.

`xmlparser.StartNamespaceDeclHandler` (*prefix*, *uri*)

Called when an element contains a namespace declaration. Namespace declarations are processed before the `StartElementHandler` is called for the element on which declarations are placed.

`xmlparser.EndNamespaceDeclHandler` (*prefix*)

Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the `StartNamespaceDeclHandler` was called to indicate the start of each namespace declaration's scope. Calls to this handler are made after the corresponding `EndElementHandler` for the end of the element.

`xmlparser.CommentHandler` (*data*)

Called for comments. *data* is the text of the comment, excluding the leading '`<!--`' and trailing '`-->`'.

`xmlparser.StartCdataSectionHandler` ()

Called at the start of a CDATA section. This and `EndCdataSectionHandler` are needed to be able to identify the syntactical start and end for CDATA sections.

`xmlparser.EndCdataSectionHandler` ()

Called at the end of a CDATA section.

`xmlparser.DefaultHandler` (*data*)

Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

`xmlparser.DefaultHandlerExpand` (*data*)

This is the same as the `DefaultHandler()`, but doesn't inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

`xmlparser.NotStandaloneHandler` ()

Called if the XML document hasn't been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set `standalone` to `yes` in an XML declaration. If this handler returns 0, then the parser will raise an `XML_ERROR_NOT_STANDALONE` error. If this handler is not set, no exception is raised by the parser for this condition.

`xmlparser.ExternalEntityRefHandler` (*context*, *base*, *systemId*, *publicId*)

Called for references to external entities. *base* is the current base, as set by a previous call to `SetBase()`. The public and system identifiers, *systemId* and *publicId*, are strings if given; if the public identifier is not given, *publicId* will be `None`. The *context* value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using `ExternalEntityParserCreate(context)`, initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns 0, the parser will raise an `XML_ERROR_EXTERNAL_ENTITY_HANDLING` error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the `DefaultHandler` callback, if provided.

20.13.2 ExpatError Exceptions

ExpatError exceptions have a number of interesting attributes :

ExpatError.code

Expat's internal error number for the specific error. The *errors.messages* dictionary maps these error numbers to Expat's error messages. For example :

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

The *errors* module also provides error message constants and a dictionary *codes* mapping these messages back to the error codes, see below.

ExpatError.lineno

Line number on which the error was detected. The first line is numbered 1.

ExpatError.offset

Character offset into the line where the error occurred. The first column is numbered 0.

20.13.3 Exemple

The following program defines three handlers that just print out their arguments.

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

The output from this program is :

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
```

(suite sur la page suivante)

(suite de la page précédente)

```
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

20.13.4 Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values : the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the `xml.parsers.expat.model` module. These constants can be collected in two groups : the model type group and the quantifier group.

The constants in the model type group are :

`xml.parsers.expat.model.XML_CTYPE_ANY`

The element named by the model name was declared to have a content model of ANY.

`xml.parsers.expat.model.XML_CTYPE_CHOICE`

The named element allows a choice from a number of options ; this is used for content models such as (A | B | C).

`xml.parsers.expat.model.XML_CTYPE_EMPTY`

Elements which are declared to be EMPTY have this model type.

`xml.parsers.expat.model.XML_CTYPE_MIXED`

`xml.parsers.expat.model.XML_CTYPE_NAME`

`xml.parsers.expat.model.XML_CTYPE_SEQ`

Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as (A, B, C).

The constants in the quantifier group are :

`xml.parsers.expat.model.XML_CQUANT_NONE`

No modifier is given, so it can appear exactly once, as for A.

`xml.parsers.expat.model.XML_CQUANT_OPT`

The model is optional : it can appear once or not at all, as for A?.

`xml.parsers.expat.model.XML_CQUANT_PLUS`

The model must occur one or more times (like A+).

`xml.parsers.expat.model.XML_CQUANT_REP`

The model must occur zero or more times, as for A*.

20.13.5 Expat error constants

The following constants are provided in the `xml.parsers.expat.errors` module. These constants are useful in interpreting some of the attributes of the `ExpatriError` exception objects raised when an error has occurred. Since for backwards compatibility reasons, the constants' value is the error *message* and not the numeric error *code*, you do this by comparing its `code` attribute with `errors.codes[errors.XML_ERROR_CONSTANT_NAME]`.

The `errors` module has the following attributes :

`xml.parsers.expat.errors.codes`

A dictionary mapping numeric error codes to their string descriptions.

Nouveau dans la version 3.2.

`xml.parsers.expat.errors.messages`

A dictionary mapping string descriptions to their error codes.

Nouveau dans la version 3.2.

`xml.parsers.expat.errors.XML_ERROR_ASYNC_ENTITY`

`xml.parsers.expat.errors.XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF`

An entity reference in an attribute value referred to an external entity instead of an internal entity.

`xml.parsers.expat.errors.XML_ERROR_BAD_CHAR_REF`

A character reference referred to a character which is illegal in XML (for example, character 0, or "�").

`xml.parsers.expat.errors.XML_ERROR_BINARY_ENTITY_REF`

An entity reference referred to an entity which was declared with a notation, so cannot be parsed.

`xml.parsers.expat.errors.XML_ERROR_DUPLICATE_ATTRIBUTE`

An attribute was used more than once in a start tag.

`xml.parsers.expat.errors.XML_ERROR_INCORRECT_ENCODING`

`xml.parsers.expat.errors.XML_ERROR_INVALID_TOKEN`

Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value 0) in a UTF-8 input stream.

`xml.parsers.expat.errors.XML_ERROR_JUNK_AFTER_DOC_ELEMENT`

Something other than whitespace occurred after the document element.

`xml.parsers.expat.errors.XML_ERROR_MISPLACED_XML_PI`

An XML declaration was found somewhere other than the start of the input data.

`xml.parsers.expat.errors.XML_ERROR_NO_ELEMENTS`

The document contains no elements (XML requires all documents to contain exactly one top-level element)..

`xml.parsers.expat.errors.XML_ERROR_NO_MEMORY`

Expat was not able to allocate memory internally.

`xml.parsers.expat.errors.XML_ERROR_PARAM_ENTITY_REF`

A parameter entity reference was found where it was not allowed.

`xml.parsers.expat.errors.XML_ERROR_PARTIAL_CHAR`

An incomplete character was found in the input.

`xml.parsers.expat.errors.XML_ERROR_RECURSIVE_ENTITY_REF`

An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

`xml.parsers.expat.errors.XML_ERROR_SYNTAX`

Some unspecified syntax error was encountered.

`xml.parsers.expat.errors.XML_ERROR_TAG_MISMATCH`
An end tag did not match the innermost open start tag.

`xml.parsers.expat.errors.XML_ERROR_UNCLOSED_TOKEN`
Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

`xml.parsers.expat.errors.XML_ERROR_UNDEFINED_ENTITY`
A reference was made to an entity which was not defined.

`xml.parsers.expat.errors.XML_ERROR_UNKNOWN_ENCODING`
The document encoding is not supported by Expat.

`xml.parsers.expat.errors.XML_ERROR_UNCLOSED_CDATA_SECTION`
A CDATA marked section was not closed.

`xml.parsers.expat.errors.XML_ERROR_EXTERNAL_ENTITY_HANDLING`

`xml.parsers.expat.errors.XML_ERROR_NOT_STANDALONE`
The parser determined that the document was not « standalone » though it declared itself to be in the XML declaration, and the `NotStandaloneHandler` was set and returned 0.

`xml.parsers.expat.errors.XML_ERROR_UNEXPECTED_STATE`

`xml.parsers.expat.errors.XML_ERROR_ENTITY_DECLARED_IN_PE`

`xml.parsers.expat.errors.XML_ERROR_FEATURE_REQUIRES_XML_DTD`
An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the `xml.parsers.expat` module.

`xml.parsers.expat.errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING`
A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by `UseForeignDTD()`.

`xml.parsers.expat.errors.XML_ERROR_UNBOUND_PREFIX`
An undeclared prefix was found when namespace processing was enabled.

`xml.parsers.expat.errors.XML_ERROR_UNDECLARING_PREFIX`
The document attempted to remove the namespace declaration associated with a prefix.

`xml.parsers.expat.errors.XML_ERROR_INCOMPLETE_PE`
A parameter entity contained incomplete markup.

`xml.parsers.expat.errors.XML_ERROR_XML_DECL`
The document contained no document element at all.

`xml.parsers.expat.errors.XML_ERROR_TEXT_DECL`
There was an error parsing a text declaration in an external entity.

`xml.parsers.expat.errors.XML_ERROR_PUBLICID`
Characters were found in the public id that are not allowed.

`xml.parsers.expat.errors.XML_ERROR_SUSPENDED`
The requested operation was made on a suspended parser, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

`xml.parsers.expat.errors.XML_ERROR_NOT_SUSPENDED`
An attempt to resume the parser was made when the parser had not been suspended.

`xml.parsers.expat.errors.XML_ERROR_ABORTED`
This should not be reported to Python applications.

`xml.parsers.expat.errors.XML_ERROR_FINISHED`
The requested operation was made on a parser which was finished parsing input, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

`xml.parsers.expat.errors.XML_ERROR_SUSPEND_PE`

Notes

Les modules documentés dans ce chapitre implémentent des protocoles relatifs à Internet et à ses technologies relatives. Ils sont tous implémentés en Python. La majorité de ces modules nécessitent la présence du module `socket` lui-même dépendant du système, mais fourni sur la plupart des plateformes populaires. Voici une vue d'ensemble :

21.1 `webbrowser` — Convenient Web-browser controller

Source code : [Lib/webbrowser.py](#)

The `webbrowser` module provides a high-level interface to allow displaying Web-based documents to users. Under most circumstances, simply calling the `open()` function from this module will do the right thing.

Under Unix, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn't available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable `BROWSER` exists, it is interpreted as the `os.pathsep`-separated list of browsers to try ahead of the platform defaults. When the value of a list part contains the string `%s`, then it is interpreted as a literal browser command line to be used with the argument URL substituted for `%s`; if the part does not contain `%s`, it is simply interpreted as the name of the browser to launch.¹

For non-Unix platforms, or when a remote browser is available on Unix, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on Unix, the controlling process will launch a new browser and wait.

The script `webbrowser` can be used as a command-line interface for the module. It accepts a URL as the argument. It accepts the following optional parameters : `-n` opens the URL in a new browser window, if possible; `-t` opens the URL in a new browser page (« tab »). The options are, naturally, mutually exclusive. Usage example :

```
python -m webbrowser -t "http://www.python.org"
```

1. Executables named here without a full path will be searched in the directories given in the `PATH` environment variable.

L'exception suivante est définie :

exception `webbrowser.Error`

Exception raised when a browser control error occurs.

Les fonctions suivantes sont définies :

`webbrowser.open(url, new=0, autoraise=True)`

Display *url* using the default browser. If *new* is 0, the *url* is opened in the same browser window if possible. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page (« tab ») is opened if possible. If *autoraise* is `True`, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable).

Note that on some platforms, trying to open a filename using this function, may work and start the operating system's associated program. However, this is neither supported nor portable.

`webbrowser.open_new(url)`

Open *url* in a new window of the default browser, if possible, otherwise, open *url* in the only browser window.

`webbrowser.open_new_tab(url)`

Open *url* in a new page (« tab ») of the default browser, if possible, otherwise equivalent to `open_new()`.

`webbrowser.get(using=None)`

Return a controller object for the browser type *using*. If *using* is `None`, return a controller for a default browser appropriate to the caller's environment.

`webbrowser.register(name, constructor, instance=None)`

Register the browser type *name*. Once a browser type is registered, the `get()` function can return a controller for that browser type. If *instance* is not provided, or is `None`, *constructor* will be called without parameters to create an instance when needed. If *instance* is provided, *constructor* will never be called, and may be `None`.

This entry point is only useful if you plan to either set the `BROWSER` variable or call `get()` with a nonempty argument matching the name of a handler you declare.

A number of browser types are predefined. This table gives the type names that may be passed to the `get()` function and the corresponding instantiations for the controller classes, all defined in this module.

Type Name	Class Name	Notes
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'netscape'	Mozilla('netscape')	
'galeon'	Galeon('galeon')	
'epiphany'	Galeon('epiphany')	
'skipstone'	BackgroundBrowser('skipstone')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'mosaic'	BackgroundBrowser('mosaic')	
'opera'	Opera()	
'grail'	Grail()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSX('default')	(3)
'safari'	MacOSX('safari')	(3)
'google-chrome'	Chrome('google-chrome')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	

Notes :

- (1) « Konqueror » is the file manager for the KDE desktop environment for Unix, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice ; the `KDEDIR` variable is not sufficient. Note also that the name « kfm » is used even when using the **konqueror** command with KDE 2 — the implementation selects the best strategy for running Konqueror.
- (2) Only on Windows platforms.
- (3) Only on Mac OS X platform.

Nouveau dans la version 3.3 : Support for Chrome/Chromium has been added.

Here are some simple examples :

```
url = 'http://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

21.1.1 Browser Controller Objects

Browser controllers provide these methods which parallel three of the module-level convenience functions :

`controller.open(url, new=0, autoraise=True)`

Display *url* using the browser handled by this controller. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page (« tab ») is opened if possible.

`controller.open_new(url)`

Open *url* in a new window of the browser handled by this controller, if possible, otherwise, open *url* in the only browser window. Alias `open_new()`.

`controller.open_new_tab(url)`

Open *url* in a new page (« tab ») of the browser handled by this controller, if possible, otherwise equivalent to `open_new()`.

Notes

21.2 cgi — Common Gateway Interface support

Code source : [Lib/cgi.py](#)

Support module for Common Gateway Interface (CGI) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.

21.2.1 Introduction

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML `<FORM>` or `<ISINDEX>` element.

Most often, CGI scripts live in the server's special `cgi-bin` directory. The HTTP server places all sorts of information about the request (such as the client's hostname, the requested URL, the query string, and lots of other goodies) in the script's shell environment, executes the script, and sends the script's output back to the client.

The script's input is connected to the client too, and sometimes the form data is read this way ; at other times the form data is passed via the « query string » part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this :

```
print("Content-Type: text/html")    # HTML is following
print()                           # blank line, end of headers
```

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here's Python code that prints a simple piece of HTML :

```
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
```

21.2.2 Using the cgi module

Begin by writing `import cgi`.

When you write a new script, consider adding these lines :

```
import cgitb
cgitb.enable()
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you'd rather not show the guts of your program to users of your script, you can have the reports saved to files instead, with code like this :

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

It's very helpful to use this feature during script development. The reports produced by `cgitb` provide information that can save you a lot of time in tracking down bugs. You can always remove the `cgitb` line later when you have tested your script and are confident that it works correctly.

To get at submitted form data, use the `FieldStorage` class. If the form contains non-ASCII characters, use the *encoding* keyword parameter set to the value of the encoding defined for the document. It is usually contained in the `META` tag in the `HEAD` section of the HTML document or by the *Content-Type* header). This reads the form contents from the standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.

The `FieldStorage` instance can be indexed like a Python dictionary. It allows membership testing with the `in` operator, and also supports the standard dictionary method `keys()` and the built-in function `len()`. Form fields containing empty strings are ignored and do not appear in the dictionary; to keep such values, provide a true value for the optional *keep_blank_values* keyword parameter when creating the `FieldStorage` instance.

For instance, the following code (which assumes that the *Content-Type* header and blank line have already been printed) checks that the fields `name` and `addr` are both set to a non-empty string :

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print("<H1>Error</H1>")
    print("Please fill in the name and addr fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
...further form processing here...
```

Here the fields, accessed through `form[key]`, are themselves instances of `FieldStorage` (or `MiniFieldStorage`, depending on the form encoding). The `value` attribute of the instance yields the string value of the field. The `getvalue()` method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by `form[key]` is not a `FieldStorage` or `MiniFieldStorage` instance but a list of such instances. Similarly, in this situation, `form.getvalue(key)` would return a list of strings. If you expect this possibility (when your HTML form contains multiple fields with the same name), use the `getlist()` method, which always returns a list of values (so that you do not need to special-case the single item case). For example, this code concatenates any number of username fields, separated by commas :

```
value = form.getlist("username")
usernames = ",".join(value)
```

If a field represents an uploaded file, accessing the value via the `value` attribute or the `getvalue()` method reads the entire file in memory as bytes. This may not be what you want. You can test for an uploaded file by testing either the `filename` attribute or the `file` attribute. You can then read the data from the `file` attribute before it is automatically closed as part of the garbage collection of the `FieldStorage` instance (the `read()` and `readline()` methods will return bytes) :

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

`FieldStorage` objects also support being used in a `with` statement, which will automatically close them when done.

If an error is encountered when obtaining the contents of an uploaded file (for example, when the user interrupts the form submission by clicking on a Back or Cancel button) the `done` attribute of the object for the field will be set to the value `-1`.

The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive `multipart/*` encoding). When this occurs, the item will be a dictionary-like `FieldStorage` item. This can be determined by testing its `type` attribute, which should be `multipart/form-data` (or perhaps another MIME type matching `multipart/*`). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the « old » format (as the query string or as a single data part of type `application/x-www-form-urlencoded`), the items will actually be instances of the class `MiniFieldStorage`. In this case, the `list`, `file`, and `filename` attributes are always `None`.

A form submitted via POST that also has a query string will contain both `FieldStorage` and `MiniFieldStorage` items.

Modifié dans la version 3.4 : The `file` attribute is automatically closed upon the garbage collection of the creating `FieldStorage` instance.

Modifié dans la version 3.5 : Added support for the context management protocol to the `FieldStorage` class.

21.2.3 Higher Level Interface

The previous section explains how to read CGI form data using the `FieldStorage` class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn't make the techniques described in previous sections obsolete — they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name :

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name :


```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

In most situations, however, there's only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code :

```
user = form.getvalue("user").upper()
```

The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another `user=foo` pair to the query string, then the script would crash, because in this situation the `getvalue("user")` method call returns a list instead of a string. Calling the `upper()` method on a list is not valid (since lists do not have a method of this name) and results in an `AttributeError` exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That's annoying and leads to less readable scripts.

A more convenient approach is to use the methods `getfirst()` and `getlist()` provided by this higher level interface.

`FieldStorage.getfirst(name, default=None)`

This method always returns only one value associated with form field *name*. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on.¹ If no such form field or value exists then the method returns the value specified by the optional parameter *default*. This parameter defaults to `None` if not specified.

`FieldStorage.getlist(name)`

This method always returns a list of values associated with form field *name*. The method returns an empty list if no such form field or value exists for *name*. It returns a list consisting of one item if only one such value exists.

Using these methods you can write nice compact code :

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

21.2.4 Fonctions

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

`cgi.parse(fp=None, environ=os.environ, keep_blank_values=False, strict_parsing=False)`

Parse a query in the environment or from a file (the file defaults to `sys.stdin`). The `keep_blank_values` and `strict_parsing` parameters are passed to `urllib.parse.parse_qs()` unchanged.

`cgi.parse_qs(qs, keep_blank_values=False, strict_parsing=False)`

This function is deprecated in this module. Use `urllib.parse.parse_qs()` instead. It is maintained here only for backward compatibility.

`cgi.parse_qs1(qs, keep_blank_values=False, strict_parsing=False)`

This function is deprecated in this module. Use `urllib.parse.parse_qs1()` instead. It is maintained here only for backward compatibility.

1. Note that some recent versions of the HTML specification do state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is tedious and error-prone.

`cgi.parse_multipart(fp, pdict)`

Parse input of type *multipart/form-data* (for file uploads). Arguments are *fp* for the input file and *pdict* for a dictionary containing other parameters in the *Content-Type* header.

Returns a dictionary just like `urllib.parse.parse_qs()` keys are the field names, each value is a list of values for that field. This is easy to use but not much good if you are expecting megabytes to be uploaded — in that case, use the `FieldStorage` class instead which is much more flexible.

Note that this does not parse nested multipart parts — use `FieldStorage` for that.

`cgi.parse_header(string)`

Parse a MIME header (such as *Content-Type*) into a main value and a dictionary of parameters.

`cgi.test()`

Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

`cgi.print_environ()`

Format the shell environment in HTML.

`cgi.print_form(form)`

Format a form in HTML.

`cgi.print_directory()`

Format the current directory in HTML.

`cgi.print_environ_usage()`

Print a list of useful (used by CGI) environment variables in HTML.

`cgi.escape(s, quote=False)`

Convert the characters '&', '<' and '>' in string *s* to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag *quote* is true, the quotation mark character (") is also translated; this helps for inclusion in an HTML attribute value delimited by double quotes, as in ``. Note that single quotes are never translated.

Obsolète depuis la version 3.2 : This function is unsafe because *quote* is false by default, and therefore deprecated. Use `html.escape()` instead.

21.2.5 Caring about security

There's one important rule : if you invoke an external program (via the `os.system()` or `os.popen()` functions, or others with similar functionality), make very sure you don't pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn't have to come from your form !

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.

21.2.6 Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed ; usually this is in a directory `cgi-bin` in the server tree.

Make sure that your script is readable and executable by « others » ; the Unix file mode should be `00755` octal (use `chmod 0755 filename`). Make sure that the first line of the script contains `#!` starting in column 1 followed by the pathname of the Python interpreter, for instance :

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by « others ».

Make sure that any files your script needs to read or write are readable or writable, respectively, by « others » — their mode should be 00644 for readable and 00666 for writable. This is because, for security reasons, the HTTP server executes your script as user « nobody », without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server's cgi-bin directory) and the set of environment variables is also different from what you get when you log in. In particular, don't count on the shell's search path for executables (PATH) or the Python module search path (PYTHONPATH) to be set to anything interesting.

If you need to load modules from a directory which is not on Python's default module search path, you can change the path in your script, before importing other modules. For example :

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first !)

Instructions for non-Unix systems will vary; check your HTTP server's documentation (it will usually have a section on CGI scripts).

21.2.7 Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There's one reason why you should still test your script from the command line : if it contains a syntax error, the Python interpreter won't execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.

21.2.8 Debugging CGI scripts

First of all, check for trivial installation errors — reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (`cgi.py`) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it's installed in the standard `cgi-bin` directory, it should be possible to send it a request by entering a URL into your browser of the form :

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

If this gives an error of type 404, the server cannot find the script — perhaps you need to install it in a different directory. If it gives another error, there's an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as « addr » with value « At Home » and « name » with value « Joe Blow »), the `cgi.py` script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the `cgi` module's `test()` function from your script : replace its main code with the single statement

```
cgi.test()
```

This should produce the same results as those gotten from installing the `cgi.py` file itself.

When an ordinary Python script raises an unhandled exception (for whatever reason : of a typo in a module name, a file that can't be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server's log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute *some* code, you can easily send tracebacks to the Web browser using the `cgitb` module. If you haven't done so already, just add the lines :

```
import cgitb
cgitb.enable()
```

to the top of your script. Then try running it again ; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the `cgitb` module, you can use an even more robust approach (which only uses built-in modules) :

```
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will be readable.

21.2.9 Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client's display while the script is running.
- Check the installation instructions above.
- Check the HTTP server's log files. (`tail -f logfile` in a separate window may be useful !)
- Always check a script for syntax errors first, by doing something like `python script.py`.
- If your script does not have any syntax errors, try adding `import cgitb; cgitb.enable()` to the top of the script.
- When invoking external programs, make sure they can be found. Usually, this means using absolute path names
 - `PATH` is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure they can be read or written by the `userid` under which your CGI script will be running : this is typically the `userid` under which the web server is running, or some explicitly specified `userid` for a web server's `suexec` feature.
- Don't try to give a CGI script a `set-uid` mode. This doesn't work on most systems, and is a security liability as well.

Notes

21.3 `cgitb` — Gestionnaire d'exceptions pour les scripts CGI

Code source : [Lib/cgitb.py](#)

Le module `cgitb` fournit un gestionnaire d'exceptions spécifique pour les scripts Python. (Son nom est trompeur : Il a été conçu à l'origine pour afficher des piles d'appels en HTML pour les scripts CGI, puis a été généralisé par la suite pour afficher cette information en texte brut.) Une fois ce module activé, si une exception remonte jusqu'à l'interpréteur, un rapport détaillé sera affiché. Le rapport affiche la pile d'appels, montrant des extraits de code pour chaque niveau, ainsi que les arguments et les variables locales des fonctions appelantes pour vous aider à résoudre le problème. Il est aussi possible de sauvegarder cette information dans un fichier plutôt que de l'envoyer dans le navigateur.

Pour activer cette fonctionnalité, ajoutez simplement ceci au début de votre script CGI :

```
import cgitb
cgitb.enable()
```

Les paramètres optionnels de la fonction `enable()` permettent de choisir si le rapport est envoyé au navigateur ou si le rapport est écrit dans un fichier pour analyse ultérieure.

`cgitb.enable` (*display*=1, *logdir*=None, *context*=5, *format*="html")

Appeler cette fonction remplace le gestionnaire d'exceptions par défaut de l'interpréteur par celui du module `cgitb`, en configurant `sys.excepthook`.

Le paramètre optionnel *display* vaut 1 par défaut, et peut être mis à 0 pour désactiver l'envoi des piles d'appels au navigateur. Si l'argument *logdir* est donné les piles d'appels seront écrites dans des fichiers placés dans le dossier *logdir*. L'argument optionnel *context* est le nombre de lignes de code à afficher autour de la ligne courante dans le code source à chaque niveau de la pile d'appel, il vaut 5 par défaut. Si l'argument optionnel *format* est "html", le rapport sera rédigé en HTML. Le rapport sera écrit en texte brut pour toute autre valeur. La valeur par défaut est "html".

`cgitb.handler` (*info*=None)

Cette fonction gère les exceptions en utilisant la configuration par défaut (c'est à dire envoyer un rapport HTML au navigateur sans l'enregistrer dans un fichier). Il peut être utilisé lorsque vous avez attrapé une exception et que vous en voulez un rapport généré par `cgitb`. L'argument optionnel *info* doit être un *tuple* de trois éléments contenant le type de l'exception, l'exception, et la pile d'appels, tel que le *tuple* renvoyé par `sys.exc_info()`. Si l'argument *info* n'est pas donné, l'exception courante est obtenue via `sys.exc_info()`.

21.4 `wsgiref` — Outils et implémentation de référence de WSGI

WSGI (*Web Server Gateway Interface*) est une interface standard entre le serveur web et une application web écrite en Python. Avoir une interface standardisée permet de faciliter l'usage de ces applications avec un certain nombre de serveurs web différents.

Seules les personnes programmant des serveurs web et des cadriciels ont besoin de connaître les détails d'implémentation et les cas particuliers de l'architecture de WSGI. En tant qu'utilisateur WSGI vous avez uniquement besoin d'installer WSGI ou d'utiliser un cadriciel existant.

`wsgiref` est une implémentation de référence de la spécification WSGI qui peut être utilisée pour ajouter la prise en charge de WSGI par un serveur web ou par un cadriciel. Elle fournit des outils pour manipuler les variables d'environnement WSGI, les en-têtes de réponse, les classes de base pour implémenter des serveurs WSGI, un serveur de

démonstration d'application WSGI et un outil de validation qui vérifie que les serveurs et les applications WSGI sont conformes à la spécification WSGI ([PEP 3333](#)).

See <https://wsgi.readthedocs.org/> for more information about WSGI, and links to tutorials and other resources.

21.4.1 `wsgiref.util` — outils pour les environnements WSGI

Ce module fournit un certain nombre de fonctions pour manipuler des environnements WSGI. Un environnement WSGI est un dictionnaire contenant les variables de la requête HTTP comme décrit dans la [PEP 3333](#). Toutes les fonctions ayant comme argument *environ* s'attendent à ce qu'un dictionnaire compatible WSGI soit fourni ; voir la [PEP 3333](#) pour la spécification détaillée.

`wsgiref.util.guess_scheme(environ)`

Tente de déterminer s'il faut assigner « http » ou « https » à `wsgi.url_scheme`, en vérifiant si une variable d'environnement HTTPS est dans le dictionnaire *environ*. La valeur renvoyée est une chaîne de caractères.

This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a HTTPS variable with a value of « 1 » « yes », or « on » when a request is received via SSL. So, this function returns « https » if such a value is found, and « http » otherwise.

`wsgiref.util.request_uri(environ, include_query=True)`

Return the full request URI, optionally including the query string, using the algorithm found in the « URL Re-construction » section of [PEP 3333](#). If *include_query* is false, the query string is not included in the resulting URI.

`wsgiref.util.application_uri(environ)`

Similar to `request_uri()`, except that the `PATH_INFO` and `QUERY_STRING` variables are ignored. The result is the base URI of the application object addressed by the request.

`wsgiref.util.shift_path_info(environ)`

Shift a single name from `PATH_INFO` to `SCRIPT_NAME` and return the name. The *environ* dictionary is *modified* in-place; use a copy if you need to keep the original `PATH_INFO` or `SCRIPT_NAME` intact.

If there are no remaining path segments in `PATH_INFO`, None is returned.

Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at `/foo`, and the request URI path is `/foo/bar/baz`, and the WSGI application at `/foo` calls `shift_path_info()`, it will receive the string « bar », and the environment will be updated to be suitable for passing to a WSGI application at `/foo/bar`. That is, `SCRIPT_NAME` will change from `/foo` to `/foo/bar`, and `PATH_INFO` will change from `/bar/baz` to `/baz`.

When `PATH_INFO` is just a « / », this routine returns an empty string and appends a trailing slash to `SCRIPT_NAME`, even though empty path segments are normally ignored, and `SCRIPT_NAME` doesn't normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in `/x` from ones ending in `/x/` when using this routine to do object traversal.

`wsgiref.util.setup_testing_defaults(environ)`

Met à jour *environ* avec des valeurs par défaut pour des cas de tests.

Cette fonction ajoute des paramètres requis pour WSGI, en particulier `HTTP_HOST`, `SERVER_NAME`, `SERVER_PORT`, `REQUEST_METHOD`, `SCRIPT_NAME`, `PATH_INFO` et toutes les autres variables WSGI définies dans la [PEP 3333](#). Elle ne fournit que des valeurs par défaut sans toucher aux valeurs déjà définies de ces variables.

Cette fonction a pour but de faciliter les tests unitaires des serveurs et des applications WSGI dans des environnements factices. Elle ne devrait pas être utilisée dans une application ou un serveur WSGI, étant donné que les données sont factices !

Exemple d'utilisation :

```

from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
            for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()

```

In addition to the environment functions above, the `wsgiref.util` module also provides these miscellaneous utilities :

`wsgiref.util.is_hop_by_hop(header_name)`

Return true if “header_name” is an HTTP/1.1 « Hop-by-Hop » header, as defined by [RFC 2616](#).

class `wsgiref.util.FileWrapper` (*filelike*, *blksize=8192*)

A wrapper to convert a file-like object to an *iterator*. The resulting objects support both `__getitem__()` and `__iter__()` iteration styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional *blksize* parameter will be repeatedly passed to the *filelike* object’s `read()` method to obtain bytestrings to yield. When `read()` returns an empty bytestring, iteration is ended and is not resumable.

If *filelike* has a `close()` method, the returned object will also have a `close()` method, and it will invoke the *filelike* object’s `close()` method when called.

Exemple d’utilisation :

```

from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)

```


21.4.2 wsgiref.headers – WSGI response header tools

This module provides a single class, *Headers*, for convenient manipulation of WSGI response headers using a mapping-like interface.

class wsgiref.headers.**Headers** ([*headers*])

Create a mapping-like object wrapping *headers*, which must be a list of header name/value tuples as described in [PEP 3333](#). The default value of *headers* is an empty list.

Headers objects support typical mapping operations including `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()` and `__contains__()`. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, *Headers* objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns `None`, and deleting a nonexistent header does nothing.

Headers objects also support `keys()`, `values()`, and `items()` methods. The lists returned by `keys()` and `items()` can include the same key more than once if there is a multi-valued header. The `len()` of a *Headers* object is the same as the length of its `items()`, which is the same as the length of the wrapped header list. In fact, the `items()` method just returns a copy of the wrapped header list.

Calling `bytes()` on a *Headers* object returns a formatted bytestring suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the bytestring is terminated with a blank line.

In addition to their mapping interface and formatting features, *Headers* objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters :

get_all (*name*)

Renvoie une liste de toutes les valeurs pour l'en-tête *name*.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header (*name*, *value*, ***_params*)

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or `None`. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form `name="value"`. If it is `None`, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage :

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

Le code ci-dessus ajoute un en-tête qui ressemble à ceci :

```
Content-Disposition: attachment; filename="bud.gif"
```

Modifié dans la version 3.5 : Le paramètre *headers* est optionnel.

21.4.3 `wsgiref.simple_server` – a simple WSGI HTTP server

This module implements a simple HTTP server (based on `http.server`) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses `PATH_INFO` to select which application to invoke for each request. (E.g., using the `shift_path_info()` function from `wsgiref.util`.)

`wsgiref.simple_server.make_server(host, port, app, server_class=WSGIServer, handler_class=WSGIRequestHandler)`

Create a new WSGI server listening on *host* and *port*, accepting connections for *app*. The return value is an instance of the supplied *server_class*, and will process requests using the specified *handler_class*. *app* must be a WSGI application object, as defined by [PEP 3333](#).

Exemple d'utilisation :

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

`wsgiref.simple_server.demo_app(environ, start_response)`

This function is a small but complete WSGI application that returns a text page containing the message « Hello world! » and a list of the key/value pairs provided in the *environ* parameter. It's useful for verifying that a WSGI server (such as `wsgiref.simple_server`) is able to run a simple WSGI application correctly.

class `wsgiref.simple_server.WSGIServer(server_address, RequestHandlerClass)`

Create a `WSGIServer` instance. *server_address* should be a (host,port) tuple, and *RequestHandlerClass* should be the subclass of `http.server.BaseHTTPRequestHandler` that will be used to process requests. You do not normally need to call this constructor, as the `make_server()` function can handle all the details for you.

`WSGIServer` is a subclass of `http.server.HTTPServer`, so all of its methods (such as `serve_forever()` and `handle_request()`) are available. `WSGIServer` also provides these WSGI-specific methods :

set_app(application)

Sets the callable *application* as the WSGI application that will receive requests.

get_app()

Returns the currently-set application callable.

Normally, however, you do not need to use these additional methods, as `set_app()` is normally called by `make_server()`, and the `get_app()` exists mainly for the benefit of request handler instances.

class `wsgiref.simple_server.WSGIRequestHandler(request, client_address, server)`

Create an HTTP handler for the given *request* (i.e. a socket), *client_address* (a (host,port) tuple), and *server* (`WSGIServer` instance).

You do not need to create instances of this class directly ; they are automatically created as needed by `WSGIServer` objects. You can, however, subclass this class and supply it as a *handler_class* to the `make_server()` function. Some possibly relevant methods for overriding in subclasses :

get_environ()

Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the `WSGIServer` object's `base_environ` dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in [PEP 3333](#).

get_stderr()

Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

handle()

Process the HTTP request. The default implementation creates a handler instance using a `wsgiref.handlers` class to implement the actual WSGI application interface.

21.4.4 `wsgiref.validate` — WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using `wsgiref.validate`. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete **PEP 3333** compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the `paste.lint` module from Ian Bicking's « Python Paste » library.

wsgiref.validate.validator(application)

Wrap *application* and return a new WSGI application object. The returned application will forward all requests to the original *application*, and will check that both the *application* and the server invoking it are conforming to the WSGI specification and to **RFC 2616**.

Any detected nonconformance results in an `AssertionError` being raised; note, however, that how these errors are handled is server-dependent. For example, `wsgiref.simple_server` and other servers based on `wsgiref.handlers` (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to `sys.stderr` or some other error stream.

This wrapper may also generate output using the `warnings` module to indicate behaviors that are questionable but which may not actually be prohibited by **PEP 3333**. Unless they are suppressed using Python command-line options or the `warnings` API, any such warnings will be written to `sys.stderr` (*not* `wsgi.errors`, unless they happen to be the same object).

Exemple d'utilisation :

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK' # HTTP Status
    headers = [('Content-type', 'text/plain')] # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000...")
    httpd.serve_forever()
```

21.4.5 wsgiref.handlers – server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

class wsgiref.handlers.CGIHandler

CGI-based invocation via `sys.stdin`, `sys.stdout`, `sys.stderr` and `os.environ`. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke `CGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

This class is a subclass of `BaseCGIHandler` that sets `wsgi.run_once` to `true`, `wsgi.multithread` to `false`, and `wsgi.multiprocess` to `true`, and always uses `sys` and `os` to obtain the necessary CGI streams and environment.

class wsgiref.handlers.IISCGIHandler

A specialized alternative to `CGIHandler`, for use when deploying on Microsoft's IIS web server, without having set the config `allowPathInfo` option (IIS \geq 7) or metabase `allowPathInfoForScriptMappings` (IIS $<$ 7).

By default, IIS gives a `PATH_INFO` that duplicates the `SCRIPT_NAME` at the front, causing problems for WSGI applications that wish to implement routing. This handler strips any such duplicated path.

IIS can be configured to pass the correct `PATH_INFO`, but this causes another bug where `PATH_TRANSLATED` is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS $<$ 7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the `PATH_TRANSLATED` bug. For this reason IIS $<$ 7 is almost never deployed with the fix. (Even IIS7 rarely uses it because there is still no UI for it.)

There is no way for CGI code to tell whether the option was set, so a separate handler class is provided. It is used in the same way as `CGIHandler`, i.e., by calling `IISCGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

Nouveau dans la version 3.2.

class wsgiref.handlers.BaseCGIHandler(*stdin, stdout, stderr, environ, multithread=True, multiprocess=False*)

Similar to `CGIHandler`, but instead of using the `sys` and `os` modules, the CGI environment and I/O streams are specified explicitly. The `multithread` and `multiprocess` values are used to set the `wsgi.multithread` and `wsgi.multiprocess` flags for any applications run by the handler instance.

This class is a subclass of `SimpleHandler` intended for use with software other than HTTP « origin servers ». If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a `Status:` header to send an HTTP status, you probably want to subclass this instead of `SimpleHandler`.

class wsgiref.handlers.SimpleHandler(*stdin, stdout, stderr, environ, multithread=True, multiprocess=False*)

Similar to `BaseCGIHandler`, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of `BaseCGIHandler`.

This class is a subclass of `BaseHandler`. It overrides the `__init__()`, `get_stdin()`, `get_stderr()`, `add_cgi_vars()`, `_write()`, and `_flush()` methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the `stdin`, `stdout`, `stderr`, and `environ` attributes.

The `write()` method of `stdout` should write each chunk in full, like `io.BufferedIOBase`.

class wsgiref.handlers.BaseHandler

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

`BaseHandler` instances have only one method intended for external use :

run(*app*)

Run the specified WSGI application, *app*.

All of the other `BaseHandler` methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass :

`_write (data)`

Buffer the bytes *data* for transmission to the client. It's okay if this method actually transmits the data; *BaseHandler* just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

`_flush ()`

Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if *_write ()* actually sends the data).

`get_stdin ()`

Return an input stream object suitable for use as the *wsgi.input* of the request currently being processed.

`get_stderr ()`

Return an output stream object suitable for use as the *wsgi.errors* of the request currently being processed.

`add_cgi_vars ()`

Insert CGI variables for the current request into the *environ* attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized *BaseHandler* subclass.

Attributes and methods for customizing the WSGI environment :

`wsgi_multithread`

The value to be used for the *wsgi.multithread* environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

`wsgi_multiprocess`

The value to be used for the *wsgi.multiprocess* environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

`wsgi_run_once`

The value to be used for the *wsgi.run_once* environment variable. It defaults to false in *BaseHandler*, but *CGIHandler* sets it to true by default.

`os_environ`

The default environment variables to be included in every request's WSGI environment. By default, this is a copy of *os.environ* at the time that *wsgiref.handlers* was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

`server_software`

If the *origin_server* attribute is set, this attribute's value is used to set the default *SERVER_SOFTWARE* WSGI environment variable, and also to set a default *Server*: header in HTTP responses. It is ignored for handlers (such as *BaseCGIHandler* and *CGIHandler*) that are not HTTP origin servers.

Modifié dans la version 3.3 : The term « Python » is replaced with implementation specific term like « CPython », « Jython » etc.

`get_scheme ()`

Return the URL scheme being used for the current request. The default implementation uses the *guess_scheme ()* function from *wsgiref.util* to guess whether the scheme should be « http » or « https », based on the current request's *environ* variables.

`setup_environ ()`

Set the *environ* attribute to a fully-populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the *get_stdin ()*, *get_stderr ()*, and *add_cgi_vars ()* methods and the *wsgi_file_wrapper* attribute. It also inserts a *SERVER_SOFTWARE* key if not present, as long as the *origin_server* attribute is a true value and the *server_software* attribute is set.

Methods and attributes for customizing exception handling :

`log_exception (exc_info)`

Log the *exc_info* tuple in the server log. *exc_info* is a (type, value, traceback) tuple. The default

implementation simply writes the traceback to the request's `wsgi.errors` stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit

The maximum number of frames to include in tracebacks output by the default `log_exception()` method. If `None`, all frames are included.

error_output (*environ*, *start_response*)

This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error information using `sys.exc_info()`, and should pass that information to *start_response* when calling it (as described in the « Error Handling » section of [PEP 3333](#)).

The default implementation just uses the `error_status`, `error_headers`, and `error_body` attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it's not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn't include any.

error_status

The HTTP status used for error responses. This should be a status string as defined in [PEP 3333](#); it defaults to a 500 code and message.

error_headers

The HTTP headers used for error responses. This should be a list of WSGI response headers ((name, value) tuples), as described in [PEP 3333](#). The default list just sets the content type to `text/plain`.

error_body

The error response body. This should be an HTTP response body bytestring. It defaults to the plain text, « A server error occurred. Please contact the administrator. »

Methods and attributes for [PEP 3333](#)'s « Optional Platform-Specific File Handling » feature :

wsgi_file_wrapper

A `wsgi.file_wrapper` factory, or `None`. The default value of this attribute is the `wsgiref.util.FileWrapper` class.

sendfile()

Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the `wsgi_file_wrapper` attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed.

The default implementation of this method just returns a false value.

Miscellaneous methods and attributes :

origin_server

This attribute should be set to a true value if the handler's `_write()` and `_flush()` are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

This attribute's default value is true in `BaseHandler`, but false in `BaseCGIHandler` and `CGIHandler`.

http_version

If `origin_server` is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to `"1.0"`.

`wsgiref.handlers.read_environ()`

Transcode CGI variables from `os.environ` to PEP 3333 « bytes in unicode » strings, returning a new dictionary. This function is used by `CGIHandler` and `IISCGIHandler` in place of directly using `os.environ`, which is not necessarily WSGI-compliant on all platforms and web servers using Python 3 – specifically, ones where the OS's actual environment is Unicode (i.e. Windows), or ones where the environment is bytes, but the system encoding used by Python to decode it is anything other than ISO-8859-1 (e.g. Unix systems using UTF-8).

If you are implementing a CGI-based handler of your own, you probably want to use this routine instead of just copying values out of `os.environ` directly.

Nouveau dans la version 3.2.

21.4.6 Exemples

This is a working « Hello World » WSGI application :

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object (see PEP 333).
def hello_world_app(environ, start_response):
    status = '200 OK' # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')] # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server('', 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

21.5 urllib — Modules de gestion des URLs

Code source : [Lib/urllib/](#)

`urllib` est un paquet qui collecte plusieurs modules travaillant avec les URLs :

- `urllib.request` pour ouvrir et lire des URLs;
- `urllib.error` contenant les exceptions levées par `urllib.request`;
- `urllib.parse` pour analyser les URLs;
- `urllib.robotparser` pour analyser les fichiers `robots.txt`.

21.6 urllib.request — Extensible library for opening URLs

Source code : [Lib/urllib/request.py](#)

The `urllib.request` module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

Voir aussi :

The `Requests` package is recommended for a higher-level HTTP client interface.

The `urllib.request` module defines the following functions :


```
urllib.request.urlopen(url, data=None[, timeout], *, cafile=None, capath=None, cadefault=False,
                        context=None)
```

Open the URL *url*, which can be either a string or a *Request* object.

data must be an object specifying additional data to be sent to the server, or *None* if no such data is needed. See *Request* for details.

`urllib.request` module uses HTTP/1.1 and includes `Connection:close` header in its HTTP requests.

The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.

If *context* is specified, it must be a *ssl.SSLContext* instance describing the various SSL options. See *HTTPConnection* for more details.

The optional *cafile* and *capath* parameters specify a set of trusted CA certificates for HTTPS requests. *cafile* should point to a single file containing a bundle of CA certificates, whereas *capath* should point to a directory of hashed certificate files. More information can be found in *ssl.SSLContext.load_verify_locations()*.

The *cadefault* parameter is ignored.

This function always returns an object which can work as a *context manager* and has methods such as

- `geturl()` — return the URL of the resource retrieved, commonly used to determine if a redirect was followed
- `info()` — return the meta-information of the page, such as headers, in the form of an *email.message_from_string()* instance (see *Quick Reference to HTTP Headers*)
- `getcode()` — return the HTTP status code of the response.

For HTTP and HTTPS URLs, this function returns a *http.client.HTTPResponse* object slightly modified. In addition to the three new methods above, the *msg* attribute contains the same information as the *reason* attribute — the reason phrase returned by server — instead of the response headers as it is specified in the documentation for *HTTPResponse*.

For FTP, file, and data URLs and requests explicitly handled by legacy *URLOpener* and *FancyURLOpener* classes, this function returns a *urllib.response.addinfourl* object.

Raises *URLError* on protocol errors.

Note that *None* may be returned if no handler handles the request (though the default installed global *OpenerDirector* uses *UnknownHandler* to ensure this never happens).

In addition, if proxy settings are detected (for example, when a `*_proxy` environment variable like `http_proxy` is set), *ProxyHandler* is default installed and makes sure the requests are handled through the proxy.

The legacy `urllib.urlopen` function from Python 2.6 and earlier has been discontinued; *urllib.request.urlopen()* corresponds to the old `urllib2.urlopen`. Proxy handling, which was done by passing a dictionary parameter to `urllib.urlopen`, can be obtained by using *ProxyHandler* objects.

Modifié dans la version 3.2 : *cafile* and *capath* were added.

Modifié dans la version 3.2 : HTTPS virtual hosts are now supported if possible (that is, if *ssl.HAS_SNI* is true).

Nouveau dans la version 3.2 : *data* can be an iterable object.

Modifié dans la version 3.3 : *cadefault* was added.

Modifié dans la version 3.4.3 : *context* was added.

Obsolète depuis la version 3.6 : *cafile*, *capath* and *cadefault* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

```
urllib.request.install_opener(opener)
```

Install an *OpenerDirector* instance as the default global opener. Installing an opener is only necessary if you want `urlopen` to use that opener; otherwise, simply call *OpenerDirector.open()* instead of *urlopen()*. The code does not check for a real *OpenerDirector*, and any class with the appropriate interface will work.

```
urllib.request.build_opener([handler, ...])
```

Return an *OpenerDirector* instance, which chains the handlers in the order given. *handlers* can be either instances of *BaseHandler*, or subclasses of *BaseHandler* (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the *handlers*, unless the

handlers contain them, instances of them or subclasses of them : *ProxyHandler* (if proxy settings are detected), *UnknownHandler*, *HTTPHandler*, *HTTPDefaultErrorHandler*, *HTTPRedirectHandler*, *FTPHandler*, *FileHandler*, *HTTPErrorProcessor*.

If the Python installation has SSL support (i.e., if the *ssl* module can be imported), *HTTPSHandler* will also be added.

A *BaseHandler* subclass may also change its *handler_order* attribute to modify its position in the handlers list.

`urllib.request.pathname2url (path)`

Convert the pathname *path* from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the *quote()* function.

`urllib.request.url2pathname (path)`

Convert the path component *path* from a percent-encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses *unquote()* to decode *path*.

`urllib.request.getproxies ()`

This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named `<scheme>_proxy`, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from Mac OSX System Configuration for Mac OS X and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

Note : If the environment variable `REQUEST_METHOD` is set, which usually indicates your script is running in a CGI environment, the environment variable `HTTP_PROXY` (uppercase `_PROXY`) will be ignored. This is because that variable can be injected by a client using the « Proxy : » HTTP header. If you need to use an HTTP proxy in a CGI environment, either use *ProxyHandler* explicitly, or make sure the variable name is in lowercase (or at least the `_proxy` suffix).

The following classes are provided :

class `urllib.request.Request (url, data=None, headers={}, origin_req_host=None, unverifiable=False, method=None)`

This class is an abstraction of a URL request.

url should be a string containing a valid URL.

data must be an object specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use *data*. The supported object types include bytes, file-like objects, and iterables. If no `Content-Length` nor `Transfer-Encoding` header field has been provided, *HTTPHandler* will set these headers according to the type of *data*. `Content-Length` will be used to send bytes objects, while `Transfer-Encoding: chunked` as specified in **RFC 7230**, Section 3.3.1 will be used to send files and other iterables.

For an HTTP POST request method, *data* should be a buffer in the standard *application/x-www-form-urlencoded* format. The `urllib.parse.urlencode()` function takes a mapping or sequence of 2-tuples and returns an ASCII string in this format. It should be encoded to bytes before being used as the *data* parameter.

headers should be a dictionary, and will be treated as if *add_header()* was called with each key and value as arguments. This is often used to « spoof » the User-Agent header value, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11", while *urllib*'s default user agent string is "Python-urllib/2.6" (on Python 2.6).

An appropriate `Content-Type` header should be included if the *data* argument is present. If this header has not been provided and *data* is not `None`, `Content-Type: application/x-www-form-urlencoded` will be added as a default.

The final two arguments are only of interest for correct handling of third-party HTTP cookies :

origin_req_host should be the request-host of the origin transaction, as defined by [RFC 2965](#). It defaults to `http.cookiejar.request_host(self)`. This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by [RFC 2965](#). It defaults to `False`. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

method should be a string that indicates the HTTP request method that will be used (e.g. 'HEAD'). If provided, its value is stored in the *method* attribute and is used by `get_method()`. The default is 'GET' if *data* is None or 'POST' otherwise. Subclasses may indicate a different default method by setting the *method* attribute in the class itself.

Note : The request will not work as expected if the data object is unable to deliver its content more than once (e.g. a file or an iterable that can produce the content only once) and the request is retried for HTTP redirects or authentication. The *data* is sent to the HTTP server right away after the headers. There is no support for a 100-continue expectation in the library.

Modifié dans la version 3.3 : `Request.method` argument is added to the Request class.

Modifié dans la version 3.4 : Default `Request.method` may be indicated at the class level.

Modifié dans la version 3.6 : Do not raise an error if the Content-Length has not been provided and *data* is neither None nor a bytes object. Fall back to use chunked transfer encoding instead.

class `urllib.request.OpenerDirector`

The `OpenerDirector` class opens URLs via `BaseHandlers` chained together. It manages the chaining of handlers, and recovery from errors.

class `urllib.request.BaseHandler`

This is the base class for all registered handlers — and handles only the simple mechanics of registration.

class `urllib.request.HTTPDefaultErrorHandler`

A class which defines a default handler for HTTP error responses; all responses are turned into `HTTPError` exceptions.

class `urllib.request.HTTPRedirectHandler`

A class to handle redirections.

class `urllib.request.HTTPCookieProcessor` (*cookiejar=None*)

A class to handle HTTP Cookies.

class `urllib.request.ProxyHandler` (*proxies=None*)

Cause requests to go through a proxy. If *proxies* is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables `<protocol>_proxy`. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a Mac OS X environment proxy information is retrieved from the OS X System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

The `no_proxy` environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with `:port` appended, for example `cern.ch, ncsa.uiuc.edu, some.host:8080`.

Note : `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on `getproxies()`.

class `urllib.request.HTTPPasswordMgr`

Keep a database of (realm, uri) -> (user, password) mappings.

class urllib.request.HTTPPasswordMgrWithDefaultRealm

Keep a database of (realm, uri) -> (user, password) mappings. A realm of None is considered a catch-all realm, which is searched if no other realm fits.

class urllib.request.HTTPPasswordMgrWithPriorAuth

A variant of [HTTPPasswordMgrWithDefaultRealm](#) that also has a database of uri -> is_authenticated mappings. Can be used by a BasicAuth handler to determine when to send authentication credentials immediately instead of waiting for a 401 response first.

Nouveau dans la version 3.5.

class urllib.request.AbstractBasicAuthHandler (password_mgr=None)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with [HTTPPasswordMgr](#); refer to section [HTTPPasswordMgr Objects](#) for information on the interface that must be supported. If *password_mgr* also provides *is_authenticated* and *update_authenticated* methods (see [HTTPPasswordMgrWithPriorAuth Objects](#)), then the handler will use the *is_authenticated* result for a given URI to determine whether or not to send authentication credentials with the request. If *is_authenticated* returns True for the URI, credentials are sent. If *is_authenticated* is False, credentials are not sent, and then if a 401 response is received the request is re-sent with the authentication credentials. If authentication succeeds, *update_authenticated* is called to set *is_authenticated* True for the URI, so that subsequent requests to the URI or any of its super-URIs will automatically include the authentication credentials.

Nouveau dans la version 3.5 : Added *is_authenticated* support.

class urllib.request.HTTPBasicAuthHandler (password_mgr=None)

Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with [HTTPPasswordMgr](#); refer to section [HTTPPasswordMgr Objects](#) for information on the interface that must be supported. HTTPBasicAuthHandler will raise a *ValueError* when presented with a wrong Authentication scheme.

class urllib.request.ProxyBasicAuthHandler (password_mgr=None)

Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with [HTTPPasswordMgr](#); refer to section [HTTPPasswordMgr Objects](#) for information on the interface that must be supported.

class urllib.request.AbstractDigestAuthHandler (password_mgr=None)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with [HTTPPasswordMgr](#); refer to section [HTTPPasswordMgr Objects](#) for information on the interface that must be supported.

class urllib.request.HTTPDigestAuthHandler (password_mgr=None)

Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with [HTTPPasswordMgr](#); refer to section [HTTPPasswordMgr Objects](#) for information on the interface that must be supported. When both Digest Authentication Handler and Basic Authentication Handler are both added, Digest Authentication is always tried first. If the Digest Authentication returns a 40x response again, it is sent to Basic Authentication handler to Handle. This Handler method will raise a *ValueError* when presented with an authentication scheme other than Digest or Basic.

Modifié dans la version 3.3 : Raise *ValueError* on unsupported Authentication Scheme.

class urllib.request.ProxyDigestAuthHandler (password_mgr=None)

Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with [HTTPPasswordMgr](#); refer to section [HTTPPasswordMgr Objects](#) for information on the interface that must be supported.

class urllib.request.HTTPHandler

A class to handle opening of HTTP URLs.

class urllib.request.HTTPSHandler (debuglevel=0, context=None, check_hostname=None)

A class to handle opening of HTTPS URLs. *context* and *check_hostname* have the same meaning as in *http.client.HTTPSConnection*.

Modifié dans la version 3.2 : *context* and *check_hostname* were added.

class urllib.request.**FileHandler**

Open local files.

class urllib.request.**DataHandler**

Open data URLs.

Nouveau dans la version 3.4.

class urllib.request.**FTPHandler**

Open FTP URLs.

class urllib.request.**CacheFTPHandler**

Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib.request.**UnknownHandler**

A catch-all class to handle unknown URLs.

class urllib.request.**HTTPErrorProcessor**

Process HTTP error responses.

21.6.1 Request Objects

The following methods describe *Request*'s public interface, and so all may be overridden in subclasses. It also defines several public attributes that can be used by clients to inspect the parsed request.

Request.**full_url**

The original URL passed to the constructor.

Modifié dans la version 3.4.

Request.full_url is a property with setter, getter and a deleter. Getting *full_url* returns the original request URL with the fragment, if it was present.

Request.**type**

The URI scheme.

Request.**host**

The URI authority, typically a host, but may also contain a port separated by a colon.

Request.**origin_req_host**

The original host for the request, without port.

Request.**selector**

The URI path. If the *Request* uses a proxy, then selector will be the full URL that is passed to the proxy.

Request.**data**

The entity body for the request, or None if not specified.

Modifié dans la version 3.4 : Changing value of *Request.data* now deletes « Content-Length » header if it was previously set or calculated.

Request.**unverifiable**

boolean, indicates whether the request is unverifiable as defined by **RFC 2965**.

Request.**method**

The HTTP request method to use. By default its value is *None*, which means that *get_method()* will do its normal computation of the method to be used. Its value can be set (thus overriding the default computation in *get_method()*) either by providing a default value by setting it at the class level in a *Request* subclass, or by passing a value in to the *Request* constructor via the *method* argument.

Nouveau dans la version 3.3.

Modifié dans la version 3.4 : A default value can now be set in subclasses ; previously it could only be set via the constructor argument.

`Request.get_method()`

Return a string indicating the HTTP request method. If `Request.method` is not `None`, return its value, otherwise return 'GET' if `Request.data` is `None`, or 'POST' if it's not. This is only meaningful for HTTP requests.

Modifié dans la version 3.3 : `get_method` now looks at the value of `Request.method`.

`Request.add_header(key, val)`

Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

`Request.add_unredirected_header(key, header)`

Add a header that will not be added to a redirected request.

`Request.has_header(header)`

Return whether the instance has the named header (checks both regular and unredirected).

`Request.remove_header(header)`

Remove named header from the request instance (both from regular and unredirected headers).

Nouveau dans la version 3.4.

`Request.get_full_url()`

Return the URL given in the constructor.

Modifié dans la version 3.4.

Returns `Request.full_url`

`Request.set_proxy(host, type)`

Prepare the request by connecting to a proxy server. The *host* and *type* will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

`Request.get_header(header_name, default=None)`

Return the value of the given header. If the header is not present, return the default value.

`Request.header_items()`

Return a list of tuples (header_name, header_value) of the Request headers.

Modifié dans la version 3.4 : The request methods `add_data`, `has_data`, `get_data`, `get_type`, `get_host`, `get_selector`, `get_origin_req_host` and `is_unverifiable` that were deprecated since 3.3 have been removed.

21.6.2 OpenerDirector Objects

`OpenerDirector` instances have the following methods :

`OpenerDirector.add_handler(handler)`

handler should be an instance of `BaseHandler`. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case).

- `protocol_open()` — signal that the handler knows how to open *protocol* URLs.
- `http_error_type()` — signal that the handler knows how to handle HTTP errors with HTTP error code *type*.
- `protocol_error()` — signal that the handler knows how to handle errors from (non-http) *protocol*.
- `protocol_request()` — signal that the handler knows how to pre-process *protocol* requests.
- `protocol_response()` — signal that the handler knows how to post-process *protocol* responses.

`OpenerDirector.open(url, data=None[, timeout])`

Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of `url.open()` (which simply calls the `open()` method on the currently installed global `OpenerDirector`). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections).

`OpenerDirector.error(proto, *args)`

Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the `http_error_*()` methods of the handler classes.

Return values and exceptions raised are the same as those of `url.open()`.

OpenerDirector objects open URLs in three stages :

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like `protocol_request()` has that method called to pre-process the request.
2. Handlers with a method named like `protocol_open()` are called to handle the request. This stage ends when a handler either returns a non-*None* value (ie. a response), or raises an exception (usually *URLError*). Exceptions are allowed to propagate.

In fact, the above algorithm is first tried for methods named `default_open()`. If all such methods return *None*, the algorithm is repeated for methods named like `protocol_open()`. If all such methods return *None*, the algorithm is repeated for methods named `unknown_open()`.

Note that the implementation of these methods may involve calls of the parent `OpenerDirector` instance's `open()` and `error()` methods.

3. Every handler with a method named like `protocol_response()` has that method called to post-process the response.

21.6.3 BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use :

`BaseHandler.add_parent(director)`

Add a director as parent.

`BaseHandler.close()`

Remove any parents.

The following attribute and methods should only be used by classes derived from *BaseHandler*.

Note : The convention has been adopted that subclasses defining `protocol_request()` or `protocol_response()` methods are named **Processor*; all others are named **Handler*.

`BaseHandler.parent`

A valid *OpenerDirector*, which can be used to open using a different protocol, or handle errors.

`BaseHandler.default_open(req)`

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent *OpenerDirector*. It should return a file-like object as described in the return value of the `open()` of *OpenerDirector*, or *None*. It should raise *URLError*, unless a truly exceptional thing happens (for example, *MemoryError* should not be mapped to *URLError*).

This method will be called before any protocol-specific open method.

`BaseHandler.protocol_open(req)`

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. Return values should be the same as for `default_open()`.

`BaseHandler.unknown_open(req)`

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the *parent* `OpenerDirector`. Return values should be the same as for `default_open()`.

`BaseHandler.http_error_default(req, fp, code, msg, hdrs)`

This method is *not* defined in `BaseHandler`, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the `OpenerDirector` getting the error, and should not normally be called in other circumstances.

req will be a `Request` object, *fp* will be a file-like object with the HTTP error body, *code* will be the three-digit code of the error, *msg* will be the user-visible explanation of the code and *hdrs* will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of `urlopen()`.

`BaseHandler.http_error_nnn(req, fp, code, msg, hdrs)`

nnn should be a three-digit HTTP error code. This method is also not defined in `BaseHandler`, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code *nnn* occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for `http_error_default()`.

`BaseHandler.protocol_request(req)`

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. *req* will be a `Request` object. The return value should be a `Request` object.

`BaseHandler.protocol_response(req, response)`

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. *req* will be a `Request` object. *response* will be an object implementing the same interface as the return value of `urlopen()`. The return value should implement the same interface as the return value of `urlopen()`.

21.6.4 HTTPRedirectHandler Objects

Note : Some HTTP redirections require action from this module's client code. If this is the case, `HTTPError` is raised. See [RFC 2616](#) for details of the precise meanings of the various redirection codes.

An `HTTPError` exception raised as a security consideration if the `HTTPRedirectHandler` is presented with a redirected URL which is not an HTTP, HTTPS or FTP URL.

`HTTPRedirectHandler.redirect_request(req, fp, code, msg, hdrs, newurl)`

Return a `Request` or `None` in response to a redirect. This is called by the default implementations of the `http_error_30*()` methods when a redirection is received from the server. If a redirection should take place, return a new `Request` to allow `http_error_30*()` to perform the redirect to *newurl*. Otherwise, raise `HTTPError` if no other handler should try to handle this URL, or return `None` if you can't but another handler might.

Note : The default implementation of this method does not strictly follow [RFC 2616](#), which says that 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and the default implementation reproduces this behavior.

`HTTPRedirectHandler.http_error_301` (*req, fp, code, msg, hdrs*)

Redirect to the `Location:` or `URI:` URL. This method is called by the parent `OpenerDirector` when getting an HTTP “moved permanently” response.

`HTTPRedirectHandler.http_error_302` (*req, fp, code, msg, hdrs*)

The same as `http_error_301()`, but called for the “found” response.

`HTTPRedirectHandler.http_error_303` (*req, fp, code, msg, hdrs*)

The same as `http_error_301()`, but called for the “see other” response.

`HTTPRedirectHandler.http_error_307` (*req, fp, code, msg, hdrs*)

The same as `http_error_301()`, but called for the “temporary redirect” response.

21.6.5 HTTPCookieProcessor Objects

`HTTPCookieProcessor` instances have one attribute :

`HTTPCookieProcessor.cookiejar`

The `http.cookiejar.CookieJar` in which cookies are stored.

21.6.6 ProxyHandler Objects

`ProxyHandler.protocol_open` (*request*)

The `ProxyHandler` will have a method `protocol_open()` for every *protocol* which has a proxy in the *proxies* dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.

21.6.7 HTTPPasswordMgr Objects

These methods are available on `HTTPPasswordMgr` and `HTTPPasswordMgrWithDefaultRealm` objects.

`HTTPPasswordMgr.add_password` (*realm, uri, user, passwd*)

uri can be either a single URI, or a sequence of URIs. *realm*, *user* and *passwd* must be strings. This causes (*user*, *passwd*) to be used as authentication tokens when authentication for *realm* and a super-URI of any of the given URIs is given.

`HTTPPasswordMgr.find_user_password` (*realm, authuri*)

Get user/password for given realm and URI, if any. This method will return (`None`, `None`) if there is no matching user/password.

For `HTTPPasswordMgrWithDefaultRealm` objects, the realm `None` will be searched if the given *realm* has no matching user/password.

21.6.8 HTTPPasswordMgrWithPriorAuth Objects

This password manager extends `HTTPPasswordMgrWithDefaultRealm` to support tracking URIs for which authentication credentials should always be sent.

`HTTPPasswordMgrWithPriorAuth.add_password(realm, uri, user, passwd, is_authenticated=False)`
realm, uri, user, passwd are as for `HTTPPasswordMgr.add_password()`. *is_authenticated* sets the initial value of the *is_authenticated* flag for the given URI or list of URIs. If *is_authenticated* is specified as `True`, *realm* is ignored.

`HTTPPasswordMgr.find_user_password(realm, authuri)`
Same as for `HTTPPasswordMgrWithDefaultRealm` objects

`HTTPPasswordMgrWithPriorAuth.update_authenticated(self, uri, is_authenticated=False)`
Update the *is_authenticated* flag for the given *uri* or list of URIs.

`HTTPPasswordMgrWithPriorAuth.is_authenticated(self, authuri)`
Returns the current state of the *is_authenticated* flag for the given URI.

21.6.9 AbstractBasicAuthHandler Objects

`AbstractBasicAuthHandler.http_error_auth_reged(authreq, host, req, headers)`
Handle an authentication request by getting a user/password pair, and re-trying the request. *authreq* should be the name of the header where the information about the realm is included in the request, *host* specifies the URL and path to authenticate for, *req* should be the (failed) `Request` object, and *headers* should be the error headers.
host is either an authority (e.g. "python.org") or a URL containing an authority component (e.g. "http://python.org/"). In either case, the authority must not contain a userinfo component (so, "python.org" and "python.org:80" are fine, "joe:password@python.org" is not).

21.6.10 HTTPBasicAuthHandler Objects

`HTTPBasicAuthHandler.http_error_401(req, fp, code, msg, hdrs)`
Retry the request with authentication information, if available.

21.6.11 ProxyBasicAuthHandler Objects

`ProxyBasicAuthHandler.http_error_407(req, fp, code, msg, hdrs)`
Retry the request with authentication information, if available.

21.6.12 AbstractDigestAuthHandler Objects

`AbstractDigestAuthHandler.http_error_auth_reged(authreq, host, req, headers)`
authreq should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) `Request` object, and *headers* should be the error headers.

21.6.13 HTTPDigestAuthHandler Objects

`HTTPDigestAuthHandler.http_error_401` (*req, fp, code, msg, hdrs*)
Retry the request with authentication information, if available.

21.6.14 ProxyDigestAuthHandler Objects

`ProxyDigestAuthHandler.http_error_407` (*req, fp, code, msg, hdrs*)
Retry the request with authentication information, if available.

21.6.15 HTTPHandler Objects

`HTTPHandler.http_open` (*req*)
Send an HTTP request, which can be either GET or POST, depending on `req.has_data()`.

21.6.16 HTTPSHandler Objects

`HTTPSHandler.https_open` (*req*)
Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.

21.6.17 FileHandler Objects

`FileHandler.file_open` (*req*)
Open the file locally, if there is no host name, or the host name is 'localhost'.
Modifié dans la version 3.2 : This method is applicable only for local hostnames. When a remote hostname is given, an `URLError` is raised.

21.6.18 DataHandler Objects

`DataHandler.data_open` (*req*)
Read a data URL. This kind of URL contains the content encoded in the URL itself. The data URL syntax is specified in [RFC 2397](#). This implementation ignores white spaces in base64 encoded data URLs so the URL may be wrapped in whatever source file it comes from. But even though some browsers don't mind about a missing padding at the end of a base64 encoded data URL, this implementation will raise an `ValueError` in that case.

21.6.19 FTPHandler Objects

`FTPHandler.ftp_open` (*req*)
Open the FTP file indicated by *req*. The login is always done with empty username and password.

21.6.20 CacheFTPHandler Objects

CacheFTPHandler objects are *FTPHandler* objects with the following additional methods :

`CacheFTPHandler.setTimeout(t)`
Set timeout of connections to *t* seconds.

`CacheFTPHandler.setMaxConns(m)`
Set maximum number of cached connections to *m*.

21.6.21 UnknownHandler Objects

`UnknownHandler.unknown_open()`
Raise a *URLError* exception.

21.6.22 HTTPErrorProcessor Objects

`HTTPErrorProcessor.http_response(request, response)`
Process HTTP error responses.
For 200 error codes, the response object is returned immediately.
For non-200 error codes, this simply passes the job on to the `protocol_error_code()` handler methods, via `OpenerDirector.error()`. Eventually, *HTTPDefaultErrorHandler* will raise an *HTTPError* if no other handler handles the error.

`HTTPErrorProcessor.https_response(request, response)`
Process HTTPS error responses.
The behavior is same as `http_response()`.

21.6.23 Examples

In addition to the examples below, more examples are given in `urllib-howto`.

This example gets the python.org main page and displays the first 300 bytes of it.

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

Note that `urlopen` returns a bytes object. This is because there is no way for `urlopen` to automatically determine the encoding of the byte stream it receives from the HTTP server. In general, a program will decode the returned bytes object to string once it determines or guesses the appropriate encoding.

The following W3C document, <https://www.w3.org/International/O-charset>, lists the various ways in which an (X)HTML or an XML document could have specified its encoding information.

As the python.org website uses *utf-8* encoding as specified in its meta tag, we will use the same for decoding the bytes object.

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml"
```

It is also possible to achieve the same result without using the *context manager* approach.

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml"
```

In the following example, we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                             data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

The code for the sample CGI used in the above example is :

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

Here is an example of doing a PUT request using *Request* :

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA, method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

Use of Basic HTTP Authentication :

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                        uri='https://mahler:8092/site-updates.py',
                        user='klem',
                        passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

build_opener() provides many handlers by default, including a *ProxyHandler*. By default, *ProxyHandler* uses the environment variables named `<scheme>_proxy`, where `<scheme>` is the URL scheme involved. For

example, the `http_proxy` environment variable is read to obtain the HTTP proxy's URL.

This example replaces the default *ProxyHandler* with one that uses programmatically-supplied proxy URLs, and adds proxy authorization support with *ProxyBasicAuthHandler*.

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

Adding HTTP headers :

Use the *headers* argument to the *Request* constructor, or :

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib.request.urlopen(req)
```

OpenerDirector automatically adds a *User-Agent* header to every *Request*. To change this :

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

Also, remember that a few standard headers (*Content-Length*, *Content-Type* and *Host*) are added when the *Request* is passed to *urlopen()* (or *OpenerDirector.open()*).

Here is an example session that uses the GET method to retrieve a URL containing parameters :

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
... 
```

The following example uses the POST method instead. Note that params output from *urlencode* is encoded to bytes before it is sent to *urlopen* as data :

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
... 
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings :

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
... 
```

The following example uses no proxies at all, overriding environment settings :

```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
... 
```

21.6.24 Legacy interface

The following functions and classes are ported from the Python 2 module `urllib` (as opposed to `urllib2`). They might become deprecated at some point in the future.

`urllib.request.urlretrieve(url, filename=None, reporthook=None, data=None)`

Copy a network object denoted by a URL to a local file. If the URL points to a local file, the object will not be copied unless `filename` is supplied. Return a tuple (`filename`, `headers`) where `filename` is the local file name under which the object can be found, and `headers` is whatever the `info()` method of the object returned by `urlopen()` returned (for a remote object). Exceptions are the same as for `urlopen()`.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

The following example illustrates the most common usage scenario :

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

If the `url` uses the `http:` scheme identifier, the optional `data` argument may be given to specify a POST request (normally the request type is GET). The `data` argument must be a bytes object in standard `application/x-www-form-urlencoded` format; see the `urllib.parse.urlencode()` function.

`urlretrieve()` will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a `Content-Length` header). This can occur, for example, when the download is interrupted.

The `Content-Length` is treated as a lower bound : if there's more data to read, `urlretrieve` reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no `Content-Length` header was supplied, `urlretrieve` can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

`urllib.request.urlcleanup()`

Cleans up temporary files that may have been left behind by previous calls to `urlretrieve()`.

class `urllib.request.URLopener` (`proxies=None`, `**x509`)

Obsolète depuis la version 3.3.

Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than `http:`, `ftp:`, or `file:`, you probably want to use `FancyURLopener`.

By default, the `URLopener` class sends a *User-Agent* header of `urllib/VVV`, where `VVV` is the `urllib` version number. Applications can define their own *User-Agent* header by subclassing `URLopener` or `FancyURLopener` and setting the class attribute `version` to an appropriate string value in the subclass definition.

The optional `proxies` parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is `None`, in which case environmental proxy settings will be used if present, as discussed in the definition of `urlopen()`, above.

Additional keyword parameters, collected in `x509`, may be used for authentication of the client when using the `https:` scheme. The keywords `key_file` and `cert_file` are supported to provide an SSL key and certificate; both are needed to support client authentication.

`URLopener` objects will raise an `OSError` exception if the server returns an error code.

open (*fullurl*, *data=None*)

Open *fullurl* using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, `open_unknown()` is called. The *data* argument has the same meaning as the *data* argument of `urlopen()`.

open_unknown (*fullurl*, *data=None*)

Overridable interface to open unknown URL types.

retrieve (*url*, *filename=None*, *reporthook=None*, *data=None*)

Retrieves the contents of *url* and places it in *filename*. The return value is a tuple consisting of a local filename and either an `email.message.Message` object containing the response headers (for remote URLs) or `None` (for local URLs). The caller must then open and read the contents of *filename*. If *filename* is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and *filename* is not given, the filename is the output of `tempfile.mktemp()` with a suffix that matches the suffix of the last path component of the input URL. If *reporthook* is given, it must be a function accepting three numeric parameters: A chunk number, the maximum size chunks are read in and the total size of the download (-1 if unknown). It will be called once at the start and after each chunk of data is read from the network. *reporthook* is ignored for local URLs.

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a POST request (normally the request type is GET). The *data* argument must in standard `application/x-www-form-urlencoded` format; see the `urllib.parse.urlencode()` function.

version

Variable that specifies the user agent of the opener object. To get `urllib` to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

class `urllib.request.FancyURLopener` (...)

Obsolète depuis la version 3.3.

`FancyURLopener` subclasses `URLopener` providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the *Location* header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the *maxtries* attribute, which defaults to 10.

For all other response codes, the method `http_error_default()` is called which you can override in subclasses to handle the error appropriately.

Note : According to the letter of **RFC 2616**, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and `urllib` reproduces this behaviour.

The parameters to the constructor are the same as those for `URLopener`.

Note : When performing basic authentication, a `FancyURLopener` instance calls its `prompt_user_passwd()` method. The default implementation asks the users for the required infor-

mation on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

The `FancyURLopener` class offers one additional method that should be overloaded to provide the appropriate behavior :

prompt_user_passwd (*host, realm*)

Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, (*user, password*), which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

21.6.25 urllib.request Restrictions

- Currently, only the following protocols are supported : HTTP (versions 0.9 and 1.0), FTP, local files, and data URLs.
Modifié dans la version 3.4 : Added support for data URLs.
- The caching feature of `urlretrieve()` has been disabled until someone finds the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The `urlopen()` and `urlretrieve()` functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.
- The data returned by `urlopen()` or `urlretrieve()` is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the `Content-Type` header. If the returned data is HTML, you can use the module `html.parser` to parse it.
- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the `ftplib` module, subclassing `FancyURLopener`, or changing `_url opener` to meet your needs.

21.7 urllib.response — Response classes used by urllib

The `urllib.response` module defines functions and classes which define a minimal file like interface, including `read()` and `readline()`. The typical response object is an `addinfourl` instance, which defines an `info()` method and that returns headers and a `geturl()` method that returns the url. Functions defined by this module are used internally by the `urllib.request` module.

21.8 urllib.parse — Parse URLs into components

Code source : <Lib/urllib/parse.py>

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a « relative URL » to an absolute URL given a « base URL. »

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes : file, ftp, gopher, hdl, http, https, imap, mailto, mms, news, nntp, prospero, rsync, rtsp, rtspu, sftp, shhttp, sip, sips, snews, svn, svn+ssh, telnet, wais, ws, wss.

The `urllib.parse` module defines functions that fall into two broad categories : URL parsing and URL quoting. These are covered in detail in the following sections.

21.8.1 URL Parsing

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string.

`urllib.parse.urlparse (urlstring, scheme="", allow_fragments=True)`

Parse a URL into six components, returning a 6-tuple. This corresponds to the general structure of a URL : `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example :

```
>>> from urllib.parse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

Following the syntax specifications in **RFC 1808**, `urlparse` recognizes a netloc only if it is properly introduced by `“//”`. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

The *scheme* argument gives the default addressing scheme, to be used only if the URL does not specify one. It should be the same type (text or bytes) as *urlstring*, except that the default value `' '` is always allowed, and is automatically converted to `b' '` if appropriate.

If the `allow_fragments` argument is false, fragment identifiers are not recognized. Instead, they are parsed as part of the path, parameters or query component, and `fragment` is set to the empty string in the return value.

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes :

Attribut	Index	Valeur	Value if not present
<code>scheme</code>	0	URL scheme specifier	<code>scheme</code> parameter
<code>netloc</code>	1	Network location part	empty string
<code>path</code>	2	Hierarchical path	empty string
<code>params</code>	3	Parameters for last path element	empty string
<code>query</code>	4	Query component	empty string
<code>fragment</code>	5	Fragment identifier	empty string
<code>username</code>		User name	<code>None</code>
<code>password</code>		Password	<code>None</code>
<code>hostname</code>		Host name (lower case)	<code>None</code>
<code>port</code>		Port number as integer, if present	<code>None</code>

Reading the `port` attribute will raise a `ValueError` if an invalid port is specified in the URL. See section [Structured Parse Results](#) for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a `ValueError`.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a `ValueError`. If the URL is decomposed before parsing, no error will be raised.

Modifié dans la version 3.2 : Added IPv6 URL parsing capabilities.

Modifié dans la version 3.3 : The fragment is now parsed for all URL schemes (unless `allow_fragment` is false), in accordance with [RFC 3986](#). Previously, a whitelist of schemes that support fragments existed.

Modifié dans la version 3.6 : Out-of-range port numbers now raise `ValueError`, instead of returning `None`.

Modifié dans la version 3.6.9 : Characters that affect netloc parsing under NFKC normalization will now raise `ValueError`.

`urllib.parse.parse_qs(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace', max_num_fields=None)`

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument `keep_blank_values` is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument `strict_parsing` is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional `encoding` and `errors` parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

The optional argument `max_num_fields` is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than `max_num_fields` fields read.

Use the `urllib.parse.urlencode()` function (with the `doseq` parameter set to True) to convert such dictionaries into query strings.

Modifié dans la version 3.2 : Add `encoding` and `errors` parameters.

Modifié dans la version 3.6.8 : Added `max_num_fields` parameter.

`urllib.parse.parse_qsl(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace', max_num_fields=None)`

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a list of name, value pairs.

The optional argument `keep_blank_values` is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument `strict_parsing` is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional `encoding` and `errors` parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

The optional argument `max_num_fields` is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than `max_num_fields` fields read.

Use the `urllib.parse.urlencode()` function to convert such lists of pairs into query strings.

Modifié dans la version 3.2 : Add `encoding` and `errors` parameters.

Modifié dans la version 3.6.8 : Added `max_num_fields` parameter.

`urllib.parse.urlunparse(parts)`

Construct a URL from a tuple as returned by `urlparse()`. The `parts` argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

`urllib.parse.urlsplit(urlstring, scheme="", allow_fragments=True)`

This is similar to `urlparse()`, but does not split the params from the URL. This should generally be used instead of `urlparse()` if the more recent URL syntax allowing parameters to be applied to each segment of the `path` portion of the URL (see [RFC 2396](#)) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-tuple : (addressing scheme, network location, path, query, fragment identifier). The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes :

Attribut	Index	Valeur	Value if not present
<code>scheme</code>	0	URL scheme specifier	<code>scheme</code> parameter
<code>netloc</code>	1	Network location part	empty string
<code>path</code>	2	Hierarchical path	empty string
<code>query</code>	3	Query component	empty string
<code>fragment</code>	4	Fragment identifier	empty string
<code>username</code>		User name	<code>None</code>
<code>password</code>		Password	<code>None</code>
<code>hostname</code>		Host name (lower case)	<code>None</code>
<code>port</code>		Port number as integer, if present	<code>None</code>

Reading the `port` attribute will raise a `ValueError` if an invalid port is specified in the URL. See section [Structured Parse Results](#) for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a `ValueError`.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a `ValueError`. If the URL is decomposed before parsing, no error will be raised.

Modifié dans la version 3.6 : Out-of-range port numbers now raise `ValueError`, instead of returning `None`.

Modifié dans la version 3.6.9 : Characters that affect `netloc` parsing under NFKC normalization will now raise `ValueError`.

`urllib.parse.urlunsplit(parts)`

Combine the elements of a tuple as returned by `urlsplit()` into a complete URL as a string. The `parts` argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

`urllib.parse.urljoin(base, url, allow_fragments=True)`

Construct a full (« absolute ») URL by combining a « base URL » (`base`) with another URL (`url`). Informally, this

uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example :

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The `allow_fragments` argument has the same meaning and default as for `urlparse()`.

Note : If `url` is an absolute URL (that is, starting with `//` or `scheme://`), the `url`'s host name and/or scheme will be present in the result. For example :

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the `url` with `urlsplit()` and `urlunsplit()`, removing possible `scheme` and `netloc` parts.

Modifié dans la version 3.5 : Behaviour updated to match the semantics defined in [RFC 3986](#).

`urllib.parse.urldefrag(url)`

If `url` contains a fragment identifier, return a modified version of `url` with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in `url`, return `url` unmodified and an empty string.

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes :

Attribut	Index	Valeur	Value if not present
<code>url</code>	0	URL with no fragment	empty string
<code>fragment</code>	1	Fragment identifier	empty string

See section [Structured Parse Results](#) for more information on the result object.

Modifié dans la version 3.2 : Result is a structured object rather than a simple 2-tuple.

21.8.2 Parsing ASCII Encoded Bytes

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on `bytes` and `bytearray` objects in addition to `str` objects.

If `str` data is passed in, the result will also contain only `str` data. If `bytes` or `bytearray` data is passed in, the result will contain only `bytes` data.

Attempting to mix `str` data with `bytes` or `bytearray` in a single function call will result in a `TypeError` being raised, while attempting to pass in non-ASCII byte values will trigger `UnicodeDecodeError`.

To support easier conversion of result objects between `str` and `bytes`, all return values from URL parsing functions provide either an `encode()` method (when the result contains `str` data) or a `decode()` method (when the result contains `bytes` data). The signatures of these methods match those of the corresponding `str` and `bytes` methods (except that the default encoding is `'ascii'` rather than `'utf-8'`). Each produces a value of a corresponding type that contains either `bytes` data (for `encode()` methods) or `str` data (for `decode()` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

Modifié dans la version 3.2 : URL parsing functions now accept ASCII encoded byte sequences

21.8.3 Structured Parse Results

The result objects from the `urlparse()`, `urlsplit()` and `urldefrag()` functions are subclasses of the `tuple` type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method :

`urllib.parse.SplitResult.geturl()`

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For `urldefrag()` results, only empty fragment identifiers will be removed. For `urlsplit()` and `urlparse()` results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function :

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

The following classes provide the implementations of the structured parse results when operating on `str` objects :

class `urllib.parse.DefragResult` (*url, fragment*)

Concrete class for `urldefrag()` results containing `str` data. The `encode()` method returns a `DefragResultBytes` instance.

Nouveau dans la version 3.2.

class `urllib.parse.ParseResult` (*scheme, netloc, path, params, query, fragment*)

Concrete class for `urlparse()` results containing `str` data. The `encode()` method returns a `ParseResultBytes` instance.

class `urllib.parse.SplitResult` (*scheme, netloc, path, query, fragment*)

Concrete class for `urlsplit()` results containing `str` data. The `encode()` method returns a `SplitResultBytes` instance.

The following classes provide the implementations of the parse results when operating on `bytes` or `bytearray` objects :

class `urllib.parse.DefragResultBytes` (*url, fragment*)

Concrete class for `urldefrag()` results containing `bytes` data. The `decode()` method returns a `DefragResult` instance.

Nouveau dans la version 3.2.

class `urllib.parse.ParseResultBytes` (*scheme, netloc, path, params, query, fragment*)

Concrete class for `urlparse()` results containing `bytes` data. The `decode()` method returns a `ParseResult` instance.

Nouveau dans la version 3.2.

class `urllib.parse.SplitResultBytes` (*scheme, netloc, path, query, fragment*)

Concrete class for `urlsplit()` results containing `bytes` data. The `decode()` method returns a `SplitResult` instance.

Nouveau dans la version 3.2.

21.8.4 URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

`urllib.parse.quote(string, safe='/', encoding=None, errors=None)`

Replace special characters in *string* using the `%xx` escape. Letters, digits, and the characters `'_.'` are never quoted. By default, this function is intended for quoting the path section of URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted — its default value is `'/'`.

string may be either a *str* or a *bytes*.

The optional *encoding* and *errors* parameters specify how to deal with non-ASCII characters, as accepted by the *str.encode()* method. *encoding* defaults to `'utf-8'`. *errors* defaults to `'strict'`, meaning unsupported characters raise a *UnicodeEncodeError*. *encoding* and *errors* must not be supplied if *string* is a *bytes*, or a *TypeError* is raised.

Note that `quote(string, safe, encoding, errors)` is equivalent to `quote_from_bytes(string.encode(encoding, errors), safe)`.

Example: `quote('/El Niño/')` yields `'/El%20Ni%C3%B1o/'`.

`urllib.parse.quote_plus(string, safe=' ', encoding=None, errors=None)`

Like *quote()*, but also replace spaces by plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to `'/'`.

Example: `quote_plus('/El Niño/')` yields `'%2FE1+Ni%C3%B1o%2F'`.

`urllib.parse.quote_from_bytes(bytes, safe=' ')`

Like *quote()*, but accepts a *bytes* object rather than a *str*, and does not perform string-to-bytes encoding.

Example: `quote_from_bytes(b'a&\xef')` yields `'a%26%EF'`.

`urllib.parse.unquote(string, encoding='utf-8', errors='replace')`

Replace `%xx` escapes by their single-character equivalent. The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

string must be a *str*.

encoding defaults to `'utf-8'`. *errors* defaults to `'replace'`, meaning invalid sequences are replaced by a placeholder character.

Example: `unquote('/El%20Ni%C3%B1o/')` yields `'/El Niño/'`.

`urllib.parse.unquote_plus(string, encoding='utf-8', errors='replace')`

Like *unquote()*, but also replace plus signs by spaces, as required for unquoting HTML form values.

string must be a *str*.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields `'/El Niño/'`.

`urllib.parse.unquote_to_bytes(string)`

Replace `%xx` escapes by their single-octet equivalent, and return a *bytes* object.

string may be either a *str* or a *bytes*.

If it is a *str*, unescaped non-ASCII characters in *string* are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields `b'a&\xef'`.

`urllib.parse.urlencode(query, doseq=False, safe=' ', encoding=None, errors=None, quote_via=quote_plus)`

Convert a mapping object or a sequence of two-element tuples, which may contain *str* or *bytes* objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a *data* for POST operation with the *urlopen()* function, then it should be encoded to bytes, otherwise it would result in a *TypeError*.

The resulting string is a series of *key=value* pairs separated by `'&'` characters, where both *key* and *value* are quoted using the *quote_via* function. By default, *quote_plus()* is used to quote the values, which means

spaces are quoted as a '+' character and "/" characters are encoded as %2F, which follows the standard for GET requests (`application/x-www-form-urlencoded`). An alternate function that can be passed as `quote_via` is `quote()`, which will encode spaces as %20 and not encode "/" characters. For maximum control of what is quoted, use `quote` and specify a value for `safe`.

When a sequence of two-element tuples is used as the `query` argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter `doseq` is evaluates to `True`, individual `key=value` pairs separated by '&' are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The `safe`, `encoding`, and `errors` parameters are passed down to `quote_via` (the `encoding` and `errors` parameters are only passed when a query element is a `str`).

To reverse this encoding process, `parse_qs()` and `parse_qsl()` are provided in this module to parse query strings into Python data structures.

Refer to [urllib examples](#) to find out how `urlencode` method can be used for generating query string for a URL or data for POST.

Modifié dans la version 3.2 : Query parameter supports bytes and string objects.

Nouveau dans la version 3.5 : `quote_via` parameter.

Voir aussi :

RFC 3986 - Uniform Resource Identifiers This is the current standard (STD66). Any changes to `urllib.parse` module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL's. This specifies the parsing requirements of IPv6 URLs.

RFC 2396 - Uniform Resource Identifiers (URI) : Generic Syntax Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme. Parsing requirements for mailto URL schemes.

RFC 1808 - Relative Uniform Resource Locators This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of « Abnormal Examples » which govern the treatment of border cases.

RFC 1738 - Uniform Resource Locators (URL) This specifies the formal syntax and semantics of absolute URLs.

21.9 `urllib.error` — Classes d'exceptions levées par `urllib.request`

Code source : [Lib/urllib/error.py](#)

Le module `urllib.error` définit les classes des exceptions levées par `urllib.request`. La classe de base de ces exceptions est `URLError`.

Les exceptions suivantes sont levées par `urllib.error` aux cas appropriés :

exception `urllib.error.URLError`

Les gestionnaires lèvent cette exception (ou des exceptions dérivées) quand ils rencontrent un problème. Elle est une sous-classe de `OSError`.

reason

La raison de cette erreur. Il peut s'agir d'un message textuel ou d'une autre instance d'exception.

Modifié dans la version 3.3 : `URLError` est maintenant une sous-classe de `OSError` plutôt que `IOError`.

exception `urllib.error.HTTPError`

Bien qu'étant une exception (une sous-classe de `URLError`), une `HTTPError` peut aussi fonctionner comme une valeur de retour normale et fichier-compatible (la même chose que renvoyé par `urlopen()`). Cela est utile pour gérer les erreurs HTTP exotiques, comme les requêtes d'authentification.

code

Un statut HTTP comme défini dans la [RFC 2616](#). Cette valeur numérique correspond à une valeur trouvée dans le dictionnaire des codes comme dans `http.server.BaseHTTPRequestHandler.responses`.

reason

Il s'agit habituellement d'une chaîne de caractères expliquant la raison de l'erreur.

headers

Les en-têtes de la réponse HTTP correspondant à la requête HTTP qui a causé la `HTTPError`.
Nouveau dans la version 3.4.

exception `urllib.error.ContentTooShortError` (*msg, content*)

Cette exception est levée quand la fonction `urlretrieve()` détecte que le montant des données téléchargées est inférieur au montant attendu (donné par l'en-tête `Content-Length`). L'attribut `content` stocke les données téléchargées (et supposément tronquées).

21.10 urllib.robotparser — Analyseur de fichiers *robots.txt*

Code source : [Lib/urllib/robotparser.py](#)

Ce module fournit une simple classe, `RobotFileParser`, qui permet de savoir si un *user-agent* particulier peut accéder à une URL du site web qui a publié ce fichier `robots.txt`. Pour plus de détails sur la structure des fichiers `robots.txt`, voir <http://www.robotstxt.org/orig.html>.

class `urllib.robotparser.RobotFileParser` (*url=""*)

Cette classe fournit des méthodes pour lire, analyser et répondre aux questions à propos du fichier `robots.txt` disponible à l'adresse *url*.

set_url (*url*)

Modifie l'URL référençant le fichier `robots.txt`.

read ()

Lit le fichier `robots.txt` depuis son URL et envoie le contenu à l'analyseur.

parse (*lines*)

Analyse les lignes données en argument.

can_fetch (*useragent, url*)

Renvoie `True` si *useragent* est autorisé à accéder à *url* selon les règles contenues dans le fichier `robots.txt` analysé.

mtime ()

Renvoie le temps auquel le fichier `robots.txt` a été téléchargé pour la dernière fois. Cela est utile pour des *web spiders* de longue durée qui doivent vérifier périodiquement si le fichier est mis à jour.

modified ()

Indique que le fichier `robots.txt` a été téléchargé pour la dernière fois au temps courant.

crawl_delay (*useragent*)

Renvoie la valeur du paramètre `Crawl-delay` du `robots.txt` pour le *useragent* en question. S'il n'y a pas de tel paramètre ou qu'il ne s'applique pas au *useragent* spécifié ou si l'entrée du `robots.txt` pour ce paramètre a une syntaxe invalide, renvoie `None`.

Nouveau dans la version 3.6.

request_rate (*useragent*)

Renvoie le contenu du paramètre `Request-rate` du `robots.txt` sous la forme d'un *named tuple* `RequestRate(requests, seconds)`. S'il n'y a pas de tel paramètre ou qu'il ne s'applique pas au *useragent* spécifié ou si l'entrée du `robots.txt` pour ce paramètre a une syntaxe invalide, `None` est renvoyé.

Nouveau dans la version 3.6.

L'exemple suivant présente une utilisation basique de la classe `RobotFileParser` :

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("")
6
>>> rp.can_fetch("", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("", "http://www.musi-cal.com/")
True
```

21.11 http — modules HTTP

Code source : [Lib/http/__init__.py](#)

`http` est un paquet qui rassemble plusieurs modules servant à travailler avec le protocole HTTP (*HyperText Transfer Protocol*) :

- Le module `http.client` est un client HTTP bas niveau. Pour accéder à des ressources web, utiliser le module haut niveau `urllib.request`
- Le module `http.server` contient des classes serveur HTTP basiques basées sur `socketserver`
- Le module `http.cookies` contient des utilitaires liés à la gestion d'état HTTP via les cookies
- Le module `http.cookiejar` fournit un mécanisme de persistance des cookies

`http` est aussi un module qui définit une liste de codes d'état HTTP et les messages associés par le biais de l'énumération `http.HTTPStatus` :

class `http.HTTPStatus`

Nouveau dans la version 3.5.

Sous-classe de `enum.IntEnum` qui définit un ensemble de codes d'état HTTP, messages explicatifs et descriptions complètes écrites en anglais.

Utilisation :

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
<HTTPStatus.OK: 200>
>>> HTTPStatus.OK == 200
True
>>> http.HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
```

(suite sur la page suivante)

(suite de la page précédente)

```
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS: 101>, ...]
```

21.11.1 Codes d'état HTTP

Les codes d'état disponibles (enregistrés auprès de l'IANA) dans `http.HTTPStatus` sont :

Code	Message	Détails
100	CONTINUE	HTTP/1.1 RFC 7231 , Section 6.2.1
101	SWITCHING_PROTOCOLS	HTTP/1.1 RFC 7231 , Section 6.2.2
102	PROCESSING	<i>WebDAV</i> RFC 2518 , Section 10.1
200	OK	HTTP/1.1 RFC 7231 , Section 6.3.1
201	CREATED	HTTP/1.1 RFC 7231 , Section 6.3.2
202	ACCEPTED	HTTP/1.1 RFC 7231 , Section 6.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP/1.1 RFC 7231 , Section 6.3.4
204	NO_CONTENT	HTTP/1.1 RFC 7231 , Section 6.3.5
205	RESET_CONTENT	HTTP/1.1 RFC 7231 , Section 6.3.6
206	PARTIAL_CONTENT	HTTP/1.1 RFC 7233 , Section 4.1
207	MULTI_STATUS	<i>WebDAV</i> RFC 4918 , Section 11.1
208	ALREADY_REPORTED	<i>WebDAV</i> Binding Extensions RFC 5842 , Section 7.1 (Expérimental)
226	IM_USED	Delta Encoding in HTTP RFC 3229 , Section 10.4.1
300	MULTIPLE_CHOICES	HTTP/1.1 RFC 7231 , Section 6.4.1
301	MOVED_PERMANENTLY	HTTP/1.1 RFC 7231 , Section 6.4.2
302	FOUND	HTTP/1.1 RFC 7231 , Section 6.4.3
303	SEE_OTHER	HTTP/1.1 RFC 7231 , Section 6.4.4
304	NOT_MODIFIED	HTTP/1.1 RFC 7232 , Section 4.1
305	USE_PROXY	HTTP/1.1 RFC 7231 , Section 6.4.5
307	TEMPORARY_REDIRECT	HTTP/1.1 RFC 7231 , Section 6.4.7
308	PERMANENT_REDIRECT	Permanent Redirect RFC 7238 , Section 3 (Expérimental)
400	BAD_REQUEST	HTTP/1.1 RFC 7231 , Section 6.5.1
401	UNAUTHORIZED	HTTP/1.1 Authentication RFC 7235 , Section 3.1
402	PAYMENT_REQUIRED	HTTP/1.1 RFC 7231 , Section 6.5.2
403	FORBIDDEN	HTTP/1.1 RFC 7231 , Section 6.5.3
404	NOT_FOUND	HTTP/1.1 RFC 7231 , Section 6.5.4
405	METHOD_NOT_ALLOWED	HTTP/1.1 RFC 7231 , Section 6.5.5
406	NOT_ACCEPTABLE	HTTP/1.1 RFC 7231 , Section 6.5.6
407	PROXY_AUTHENTICATION_REQUIRED	HTTP/1.1 Authentication RFC 7235 , Section 3.2
408	REQUEST_TIMEOUT	HTTP/1.1 RFC 7231 , Section 6.5.7
409	CONFLICT	HTTP/1.1 RFC 7231 , Section 6.5.8
410	GONE	HTTP/1.1 RFC 7231 , Section 6.5.9
411	LENGTH_REQUIRED	HTTP/1.1 RFC 7231 , Section 6.5.10
412	PRECONDITION_FAILED	HTTP/1.1 RFC 7232 , Section 4.2
413	REQUEST_ENTITY_TOO_LARGE	HTTP/1.1 RFC 7231 , Section 6.5.11
414	REQUEST_URI_TOO_LONG	HTTP/1.1 RFC 7231 , Section 6.5.12
415	UNSUPPORTED_MEDIA_TYPE	HTTP/1.1 RFC 7231 , Section 6.5.13
416	REQUEST_RANGE_NOT_SATISFIABLE	HTTP/1.1 Range Requests RFC 7233 , Section 4.4

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Code	Message	Détails
417	EXPECTATION_FAILED	HTTP/1.1 RFC 7231 , Section 6.5.14
422	UNPROCESSABLE_ENTITY	WebDAV RFC 4918 , Section 11.2
423	LOCKED	WebDAV RFC 4918 , Section 11.3
424	FAILED_DEPENDENCY	WebDAV RFC 4918 , Section 11.4
426	UPGRADE_REQUIRED	HTTP/1.1 RFC 7231 , Section 6.5.15
428	PRECONDITION_REQUIRED	Additional HTTP Status Codes RFC 6585
429	TOO_MANY_REQUESTS	Additional HTTP Status Codes RFC 6585
431	REQUEST_HEADER_FIELDS_TOO_LARGE	Additional HTTP Status Codes RFC 6585
500	INTERNAL_SERVER_ERROR	HTTP/1.1 RFC 7231 , Section 6.6.1
501	NOT_IMPLEMENTED	HTTP/1.1 RFC 7231 , Section 6.6.2
502	BAD_GATEWAY	HTTP/1.1 RFC 7231 , Section 6.6.3
503	SERVICE_UNAVAILABLE	HTTP/1.1 RFC 7231 , Section 6.6.4
504	GATEWAY_TIMEOUT	HTTP/1.1 RFC 7231 , Section 6.6.5
505	HTTP_VERSION_NOT_SUPPORTED	HTTP/1.1 RFC 7231 , Section 6.6.6
506	VARIANT_ALSO_NEGOTIATES	Transparent Content Negotiation in HTTP RFC 2295 , Section 8.1 (Expérimental)
507	INSUFFICIENT_STORAGE	WebDAV RFC 4918 , Section 11.5
508	LOOP_DETECTED	WebDAV Binding Extensions RFC 5842 , Section 7.2 (Expérimental)
510	NOT_EXTENDED	An HTTP Extension Framework RFC 2774 , Section 7 (Expérimental)
511	NETWORK_AUTHENTICATION_REQUIRED	Codes d'état HTTP supplémentaires RFC 6585 , Section 6

Dans le but de préserver la compatibilité descendante, les valeurs d'énumération sont aussi présentes dans le module `http.client` sous forme de constantes. Les noms de valeurs de l'énumération sont accessibles de deux manières : par exemple, le code HTTP 200 est accessible sous les noms `http.HTTPStatus.OK` et `http.client.OK`.

21.12 http.client — HTTP protocol client

Code source : [Lib/http/client.py](http://lib/http/client.py)

This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module `urllib.request` uses it to handle URLs that use HTTP and HTTPS.

Voir aussi :

The [Requests package](#) is recommended for a higher-level HTTP client interface.

Note : HTTPS support is only available if Python was compiled with SSL support (through the `ssl` module).

The module provides the following classes :

class `http.client.HTTPConnection` (*host*, *port=None* [, *timeout*], *source_address=None*)

An `HTTPConnection` instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from.

For example, the following calls all create instances that connect to the server at the same host and port :

```

>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)

```

Modifié dans la version 3.2 : *source_address* was added.

Modifié dans la version 3.4 : The *strict* parameter was removed. HTTP 0.9-style « Simple Responses » are no longer supported.

```

class http.client.HTTPSConnection(host, port=None, key_file=None, cert_file=None[, ti-
                                meout], source_address=None, *, context=None,
                                check_hostname=None)

```

A subclass of *HTTPConnection* that uses SSL for communication with secure servers. Default port is 443. If *context* is specified, it must be a *ssl.SSLContext* instance describing the various SSL options.

Please read *Security considerations* for more information on best practices.

Modifié dans la version 3.2 : *source_address*, *context* and *check_hostname* were added.

Modifié dans la version 3.2 : This class now supports HTTPS virtual hosts if possible (that is, if *ssl.HAS_SNI* is true).

Modifié dans la version 3.4 : The *strict* parameter was removed. HTTP 0.9-style « Simple Responses » are no longer supported.

Modifié dans la version 3.4.3 : This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior *ssl._create_unverified_context()* can be passed to the *context* parameter.

Obsolète depuis la version 3.6 : *key_file* and *cert_file* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

The *check_hostname* parameter is also deprecated; the *ssl.SSLContext.check_hostname* attribute of *context* should be used instead.

```

class http.client.HTTPResponse(sock, debuglevel=0, method=None, url=None)

```

Class whose instances are returned upon successful connection. Not instantiated directly by user.

Modifié dans la version 3.4 : The *strict* parameter was removed. HTTP 0.9 style « Simple Responses » are no longer supported.

The following exceptions are raised as appropriate :

```

exception http.client.HTTPException

```

The base class of the other exceptions in this module. It is a subclass of *Exception*.

```

exception http.client.NotConnected

```

A subclass of *HTTPException*.

```

exception http.client.InvalidURL

```

A subclass of *HTTPException*, raised if a port is given and is either non-numeric or empty.

```

exception http.client.UnknownProtocol

```

A subclass of *HTTPException*.

```

exception http.client.UnknownTransferEncoding

```

A subclass of *HTTPException*.

```

exception http.client.UnimplementedFileMode

```

A subclass of *HTTPException*.

```

exception http.client.IncompleteRead

```

A subclass of *HTTPException*.

```

exception http.client.ImproperConnectionState

```

A subclass of *HTTPException*.

exception `http.client.CannotSendRequest`

A subclass of `ImproperConnectionState`.

exception `http.client.CannotSendHeader`

A subclass of `ImproperConnectionState`.

exception `http.client.ResponseNotReady`

A subclass of `ImproperConnectionState`.

exception `http.client.BadStatusLine`

A subclass of `HTTPException`. Raised if a server responds with a HTTP status code that we don't understand.

exception `http.client.LineTooLong`

A subclass of `HTTPException`. Raised if an excessively long line is received in the HTTP protocol from the server.

exception `http.client.RemoteDisconnected`

A subclass of `ConnectionResetError` and `BadStatusLine`. Raised by `HTTPConnection.getresponse()` when the attempt to read the response results in no data read from the connection, indicating that the remote end has closed the connection.

Nouveau dans la version 3.5 : Previously, `BadStatusLine('')` was raised.

Les constantes définies dans ce module sont :

`http.client.HTTP_PORT`

The default port for the HTTP protocol (always 80).

`http.client.HTTPS_PORT`

The default port for the HTTPS protocol (always 443).

`http.client.responses`

This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example : `http.client.responses[http.client.NOT_FOUND]` is 'Not Found'.

See *Codes d'état HTTP* for a list of HTTP status codes that are available in this module as constants.

21.12.1 HTTPConnection Objects

`HTTPConnection` instances have the following methods :

`HTTPConnection.request(method, url, body=None, headers={}, *, encode_chunked=False)`

This will send a request to the server using the HTTP request method *method* and the selector *url*.

If *body* is specified, the specified data is sent after the headers are finished. It may be a *str*, a *bytes-like object*, an open *file object*, or an iterable of *bytes*. If *body* is a string, it is encoded as ISO-8859-1, the default for HTTP. If it is a bytes-like object, the bytes are sent as is. If it is a *file object*, the contents of the file is sent ; this file object should support at least the `read()` method. If the file object is an instance of `io.TextIOBase`, the data returned by the `read()` method will be encoded as ISO-8859-1, otherwise the data returned by `read()` is sent as is. If *body* is an iterable, the elements of the iterable are sent as is until the iterable is exhausted.

The *headers* argument should be a mapping of extra HTTP headers to send with the request.

If *headers* contains neither Content-Length nor Transfer-Encoding, but there is a request body, one of those header fields will be added automatically. If *body* is `None`, the Content-Length header is set to 0 for methods that expect a body (PUT, POST, and PATCH). If *body* is a string or a bytes-like object that is not also a *file*, the Content-Length header is set to its length. Any other type of *body* (files and iterables in general) will be chunk-encoded, and the Transfer-Encoding header will automatically be set instead of Content-Length.

The *encode_chunked* argument is only relevant if Transfer-Encoding is specified in *headers*. If *encode_chunked* is `False`, the `HTTPConnection` object assumes that all encoding is handled by the calling code. If it is `True`, the body will be chunk-encoded.

Note : Chunked transfer encoding has been added to the HTTP protocol version 1.1. Unless the HTTP server is known to handle HTTP 1.1, the caller must either specify the Content-Length, or must pass a `str` or bytes-like object that is not also a file as the body representation.

Nouveau dans la version 3.2 : `body` can now be an iterable.

Modifié dans la version 3.6 : If neither Content-Length nor Transfer-Encoding are set in *headers*, file and iterable *body* objects are now chunk-encoded. The `encode_chunked` argument was added. No attempt is made to determine the Content-Length for file objects.

`HTTPConnection.getresponse()`

Should be called after a request is sent to get the response from the server. Returns an `HTTPResponse` instance.

Note : Note that you must have read the whole response before you can send a new request to the server.

Modifié dans la version 3.5 : If a `ConnectionError` or subclass is raised, the `HTTPConnection` object will be ready to reconnect when a new request is sent.

`HTTPConnection.set_debuglevel(level)`

Set the debugging level. The default debug level is 0, meaning no debugging output is printed. Any value greater than 0 will cause all currently defined debug output to be printed to stdout. The `debuglevel` is passed to any new `HTTPResponse` objects that are created.

Nouveau dans la version 3.1.

`HTTPConnection.set_tunnel(host, port=None, headers=None)`

Set the host and the port for HTTP Connect Tunnelling. This allows running the connection through a proxy server. The host and port arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, *not* the address of the proxy server).

The headers argument should be a mapping of extra HTTP headers to send with the CONNECT request.

For example, to tunnel through a HTTPS proxy server running locally on port 8080, we would pass the address of the proxy to the `HTTPSConnection` constructor, and the address of the host that we eventually want to reach to the `set_tunnel()` method :

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD", "/index.html")
```

Nouveau dans la version 3.2.

`HTTPConnection.connect()`

Connect to the server specified when the object was created. By default, this is called automatically when making a request if the client does not already have a connection.

`HTTPConnection.close()`

Close the connection to the server.

As an alternative to using the `request()` method described above, you can also send your request step by step, by using the four functions below.

`HTTPConnection.putrequest(method, url, skip_host=False, skip_accept_encoding=False)`

This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *method* string, the *url* string, and the HTTP version (HTTP/1.1). To disable automatic sending of `Host :` or `Accept-Encoding :` headers (for example to accept additional content encodings), specify *skip_host* or *skip_accept_encoding* with non-False values.

`HTTPConnection.putheader(header, argument[, ...])`

Send an **RFC 822**-style header to the server. It sends a line to the server consisting of the header, a colon and a

space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

`HTTPConnection.endheaders (message_body=None, *, encode_chunked=False)`

Send a blank line to the server, signalling the end of the headers. The optional *message_body* argument can be used to pass a message body associated with the request.

If *encode_chunked* is `True`, the result of each iteration of *message_body* will be chunk-encoded as specified in [RFC 7230](#), Section 3.3.1. How the data is encoded is dependent on the type of *message_body*. If *message_body* implements the buffer interface the encoding will result in a single chunk. If *message_body* is a `collections.Iterable`, each iteration of *message_body* will result in a chunk. If *message_body* is a *file object*, each call to `.read()` will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after *message_body*.

Note : Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

Nouveau dans la version 3.6 : Chunked encoding support. The *encode_chunked* parameter was added.

`HTTPConnection.send (data)`

Send data to the server. This should be used directly only after the *endheaders()* method has been called and before *getresponse()* is called.

21.12.2 HTTPResponse Objects

An *HTTPResponse* instance wraps the HTTP response from the server. It provides access to the request headers and the entity body. The response is an iterable object and can be used in a `with` statement.

Modifié dans la version 3.5 : The *io.BufferedIOBase* interface is now implemented and all of its reader operations are supported.

`HTTPResponse.read ([amt])`

Reads and returns the response body, or up to the next *amt* bytes.

`HTTPResponse.readinto (b)`

Reads up to the next `len(b)` bytes of the response body into the buffer *b*. Returns the number of bytes read.

Nouveau dans la version 3.3.

`HTTPResponse.getheader (name, default=None)`

Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by “, “. If “default” is any iterable other than a single string, its elements are similarly returned joined by commas.

`HTTPResponse.getheaders ()`

Return a list of (header, value) tuples.

`HTTPResponse.fileno ()`

Return the *fileno* of the underlying socket.

`HTTPResponse.msg`

A `http.client.HTTPMessage` instance containing the response headers. `http.client.HTTPMessage` is a subclass of *email.message.Message*.

`HTTPResponse.version`

HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

`HTTPResponse.status`

Status code returned by server.

`HTTPResponse.reason`

Reason phrase returned by server.

`HTTPResponse.debuglevel`

A debugging hook. If `debuglevel` is greater than zero, messages will be printed to stdout as the response is read and parsed.

`HTTPResponse.closed`

Is True if the stream is closed.

21.12.3 Exemples

Here is an example session that uses the GET method :

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read() # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while not r1.closed:
...     print(r1.read(200)) # 200 bytes
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the HEAD method. Note that the HEAD method never returns any data.

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

Here is an example session that shows how to POST requests :

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action':
↳ 'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...           "Accept": "text/plain"}
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/
↳issue12524</a>'
>>> conn.close()
```

Client side HTTP PUT requests are very similar to POST requests. The difference lies only the server side where HTTP server will allow resources to be created via PUT request. It should be noted that custom HTTP methods are also handled in `urllib.request.Request` by sending the appropriate `method` attribute. Here is an example session that shows how to do PUT request using `http.client` :

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = """filecontents"""
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

21.12.4 HTTPMessage Objects

An `http.client.HTTPMessage` instance holds the headers from an HTTP response. It is implemented using the `email.message.Message` class.

21.13 ftplib — FTP protocol client

Code source : [Lib/ftplib.py](#)

This module defines the class `FTP` and a few related items. The `FTP` class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module `urllib.request` to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet [RFC 959](#).

Here's a sample session using the `ftplib` module :

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')           # connect to host, default port
>>> ftp.login()                           # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')                     # change into "debian" directory
>>> ftp.retrlines('LIST')                 # list directory contents
-rw-rw-r-- 1 1176 1176 1063 Jun 15 10:18 README
```

(suite sur la page suivante)

(suite de la page précédente)

```

...
drwxr-sr-x   5 1176      1176      4096 Dec 19  2000 pool
drwxr-sr-x   4 1176      1176      4096 Nov 17  2008 project
drwxr-xr-x   3 1176      1176      4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()

```

Le module définit les éléments suivants :

class `ftplib.FTP` (*host=""*, *user=""*, *passwd=""*, *acct=""*, *timeout=None*, *source_address=None*)

Return a new instance of the `FTP` class. When *host* is given, the method call `connect(host)` is made. When *user* is given, additionally the method call `login(user, passwd, acct)` is made (where *passwd* and *acct* default to the empty string when not given). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used). *source_address* is a 2-tuple (*host*, *port*) for the socket to bind to as its source address before connecting.

The `FTP` class supports the `with` statement, e.g. :

```

>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp      154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp      154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp      4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp      18 Jul 10  2008 Fedora
>>>

```

Modifié dans la version 3.2 : La prise en charge de l'instruction `with` a été ajoutée.

Modifié dans la version 3.3 : *source_address* parameter was added.

class `ftplib.FTP_TLS` (*host=""*, *user=""*, *passwd=""*, *acct=""*, *keyfile=None*, *certfile=None*, *context=None*, *timeout=None*, *source_address=None*)

A `FTP` subclass which adds TLS support to FTP as described in [RFC 4217](#). Connect as usual to port 21 implicitly securing the FTP control connection before authenticating. Securing the data connection requires the user to explicitly ask for it by calling the `prot_p()` method. *context* is a `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read [Security considerations](#) for best practices.

keyfile and *certfile* are a legacy alternative to *context* – they can point to PEM-formatted private key and certificate chain files (respectively) for the SSL connection.

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : *source_address* parameter was added.

Modifié dans la version 3.4 : The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

Obsolète depuis la version 3.6 : *keyfile* and *certfile* are deprecated in favor of *context*. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system's trusted CA certificates for you.

Here's a sample session using the `FTP_TLS` class :

```

>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'

```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> ftps.prot_p()
'200 Data protection level set to "private"
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-jedi
→', 'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu', 'ignore',
→'libpuzzle', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-user-variables',
→'php-jenkins-hash', 'php-skein-hash', 'php-webdav', 'phpaudit', 'phpbench',
→'pincaster', 'ping', 'posto', 'pub', 'public', 'public_keys', 'pure-ftpd',
→'qscan', 'qtc', 'sharedance', 'skycache', 'sound', 'tmp', 'ucarp']
```

exception `ftplib.error_reply`

Exception raised when an unexpected reply is received from the server.

exception `ftplib.error_temp`

Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

exception `ftplib.error_perm`

Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

exception `ftplib.error_proto`

Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.

`ftplib.all_errors`The set of all exceptions (as a tuple) that methods of *FTP* instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as *OSError*.**Voir aussi :****Module *netrc*** Parser for the `.netrc` file format. The file `.netrc` is typically used by FTP clients to load user authentication information before prompting the user.

21.13.1 FTP Objects

Several methods are available in two flavors : one for handling text files and another for binary files. These are named for the command which is used followed by `lines` for the text version or `binary` for the binary version.*FTP* instances have the following methods :**`FTP.set_debuglevel` (*level*)**

Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

`FTP.connect` (*host*=", *port*=0, *timeout*=None, *source_address*=None)Connect to the given host and port. The default port number is 21, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If no *timeout* is passed, the global default timeout setting will be used. *source_address* is a 2-tuple (*host*, *port*) for the socket to bind to as its source address before connecting.Modifié dans la version 3.3 : *source_address* parameter was added.

- FTP.getwelcome()**
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)
- FTP.login(*user*='anonymous', *passwd*='', *acct*='')**
Log in as the given *user*. The *passwd* and *acct* parameters are optional and default to the empty string. If no *user* is specified, it defaults to 'anonymous'. If *user* is 'anonymous', the default *passwd* is 'anonymous@'. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in. The *acct* parameter supplies « accounting information »; few systems implement this.
- FTP.abort()**
Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.
- FTP.sendcmd(*cmd*)**
Send a simple command string to the server and return the response string.
- FTP.voidcmd(*cmd*)**
Send a simple command string to the server and handle the response. Return nothing if a response code corresponding to success (codes in the range 200–299) is received. Raise `error_reply` otherwise.
- FTP.retrbinary(*cmd*, *callback*, *blocksize*=8192, *rest*=None)**
Retrieve a file in binary transfer mode. *cmd* should be an appropriate RETR command: 'RETR filename'. The *callback* function is called for each block of data received, with a single bytes argument giving the data block. The optional *blocksize* argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer (which will also be the largest size of the data blocks passed to *callback*). A reasonable default is chosen. *rest* means the same thing as in the `transfercmd()` method.
- FTP.retrlines(*cmd*, *callback*=None)**
Retrieve a file or directory listing in ASCII transfer mode. *cmd* should be an appropriate RETR command (see `retrbinary()`) or a command such as LIST or NLST (usually just the string 'LIST'). LIST retrieves a list of files and information about those files. NLST retrieves a list of file names. The *callback* function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default *callback* prints the line to `sys.stdout`.
- FTP.set_pasv(*val*)**
Enable « passive » mode if *val* is true, otherwise disable passive mode. Passive mode is on by default.
- FTP.storbinary(*cmd*, *fp*, *blocksize*=8192, *callback*=None, *rest*=None)**
Store a file in binary transfer mode. *cmd* should be an appropriate STOR command: "STOR filename". *fp* is a *file object* (opened in binary mode) which is read until EOF using its `read()` method in blocks of size *blocksize* to provide the data to be stored. The *blocksize* argument defaults to 8192. *callback* is an optional single parameter callable that is called on each block of data after it is sent. *rest* means the same thing as in the `transfercmd()` method.
Modifié dans la version 3.2 : *rest* parameter added.
- FTP.storlines(*cmd*, *fp*, *callback*=None)**
Store a file in ASCII transfer mode. *cmd* should be an appropriate STOR command (see `storbinary()`). Lines are read until EOF from the *file object* *fp* (opened in binary mode) using its `readline()` method to provide the data to be stored. *callback* is an optional single parameter callable that is called on each line after it is sent.
- FTP.transfercmd(*cmd*, *rest*=None)**
Initiate a transfer over the data connection. If the transfer is active, send an EPRT or PORT command and the transfer command specified by *cmd*, and accept the connection. If the server is passive, send an EPSV or PASV command, connect to it, and start the transfer command. Either way, return the socket for the connection.
If optional *rest* is given, a REST command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that **RFC 959** requires only that *rest* be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The `transfercmd()` method, therefore, converts *rest* to

a string, but no check is performed on the string's contents. If the server does not recognize the `REST` command, an `error_reply` exception will be raised. If this happens, simply call `transfercmd()` without a `rest` argument.

FTP.**transfercmd** (*cmd*, *rest=None*)

Like `transfercmd()`, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, `None` will be returned as the expected size. *cmd* and *rest* means the same thing as in `transfercmd()`.

FTP.**mlsd** (*path=""*, *facts=[]*)

List a directory in a standardized format by using `MLSD` command ([RFC 3659](#)). If *path* is omitted the current directory is assumed. *facts* is a list of strings representing the type of information desired (e.g. `["type", "size", "perm"]`). Return a generator object yielding a tuple of two elements for every file found in *path*. First element is the file name, the second one is a dictionary containing facts about the file name. Content of this dictionary might be limited by the *facts* argument but server is not guaranteed to return all requested facts.

Nouveau dans la version 3.3.

FTP.**nlst** (*argument[, ...]*)

Return a list of file names as returned by the `NLST` command. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `NLST` command.

Note : If your server supports the command, `mlsd()` offers a better API.

FTP.**dir** (*argument[, ...]*)

Produce a directory listing as returned by the `LIST` command, printing it to standard output. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `LIST` command. If the last argument is a function, it is used as a *callback* function as for `retrlines()`; the default prints to `sys.stdout`. This method returns `None`.

Note : If your server supports the command, `mlsd()` offers a better API.

FTP.**rename** (*fromname*, *toname*)

Rename file *fromname* on the server to *toname*.

FTP.**delete** (*filename*)

Remove the file named *filename* from the server. If successful, returns the text of the response, otherwise raises `error_perm` on permission errors or `error_reply` on other errors.

FTP.**cwd** (*pathname*)

Set the current directory on the server.

FTP.**mkd** (*pathname*)

Create a new directory on the server.

FTP.**pwd** ()

Return the pathname of the current directory on the server.

FTP.**rmd** (*dirname*)

Remove the directory named *dirname* on the server.

FTP.**size** (*filename*)

Request the size of the file named *filename* on the server. On success, the size of the file is returned as an integer, otherwise `None` is returned. Note that the `SIZE` command is not standardized, but is supported by many common server implementations.

FTP.**quit** ()

Send a `QUIT` command to the server and close the connection. This is the « polite » way to close a connection, but it may raise an exception if the server responds with an error to the `QUIT` command. This implies a call to the `close()` method which renders the `FTP` instance useless for subsequent calls (see below).

`FTP.close()`

Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to `quit()`. After this call the `FTP` instance should not be used any more (after a call to `close()` or `quit()` you cannot reopen the connection by issuing another `login()` method).

21.13.2 FTP_TLS Objects

`FTP_TLS` class inherits from `FTP`, defining these additional objects :

`FTP_TLS.ssl_version`

The SSL version to use (defaults to `ssl.PROTOCOL_SSLv23`).

`FTP_TLS.auth()`

Set up a secure control connection by using TLS or SSL, depending on what is specified in the `ssl_version` attribute.

Modifié dans la version 3.4 : The method now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

`FTP_TLS.ccc()`

Revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports.

Nouveau dans la version 3.3.

`FTP_TLS.prot_p()`

Set up secure data connection.

`FTP_TLS.prot_c()`

Set up clear text data connection.

21.14 poplib — POP3 protocol client

Code source : [Lib/poplib.py](#)

This module defines a class, `POP3`, which encapsulates a connection to a POP3 server and implements the protocol as defined in [RFC 1939](#). The `POP3` class supports both the minimal and optional command sets from [RFC 1939](#). The `POP3` class also supports the STLS command introduced in [RFC 2595](#) to enable encrypted communication on an already established connection.

Additionally, this module provides a class `POP3_SSL`, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the `imaplib.IMAP4` class, as IMAP servers tend to be better implemented.

The `poplib` module provides two classes :

class `poplib.POP3` (*host*, *port*=`POP3_PORT`[, *timeout*])

This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If *port* is omitted, the standard POP3 port (110) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

class `poplib.POP3_SSL` (*host*, *port*=`POP3_SSL_PORT`, *keyfile*=`None`, *certfile*=`None`, *timeout*=`None`, *context*=`None`)

This is a subclass of `POP3` that connects to the server over an SSL encrypted socket. If *port* is not specified, 995,

the standard POP3-over-SSL port is used. *timeout* works as in the *POP3* constructor. *context* is an optional *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

keyfile and *certfile* are a legacy alternative to *context* - they can point to PEM-formatted private key and certificate chain files, respectively, for the SSL connection.

Modifié dans la version 3.2 : *context* parameter added.

Modifié dans la version 3.4 : The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Obsolète depuis la version 3.6 : *keyfile* and *certfile* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

One exception is defined as an attribute of the *poplib* module :

exception *poplib.error_proto*

Exception raised on any errors from this module (errors from *socket* module are not caught). The reason for the exception is passed to the constructor as a string.

Voir aussi :

Module *imaplib* The standard Python IMAP module.

Frequently Asked Questions About Fetchmail The FAQ for the **fetchmail** POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

21.14.1 POP3 Objects

All POP3 commands are represented by methods of the same name, in lower-case ; most return the response text sent by the server.

An *POP3* instance has the following methods :

POP3.set_debuglevel (*level*)

Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

POP3.getwelcome ()

Returns the greeting string sent by the POP3 server.

POP3.capa ()

Query the server's capabilities as specified in **RFC 2449**. Returns a dictionary in the form {'name': ['param'...] }.

Nouveau dans la version 3.4.

POP3.user (*username*)

Send user command, response should indicate that a password is required.

POP3.pass_ (*password*)

Send password, response includes message count and mailbox size. Note : the mailbox on the server is locked until *quit* () is called.

POP3.apop (*user*, *secret*)

Use the more secure APOP authentication to log into the POP3 server.

POP3.rpop (*user*)

Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

`POP3.stat()`

Get mailbox status. The result is a tuple of 2 integers : (message count, mailbox size).

`POP3.list([which])`

Request message list, result is in the form (response, ['mesg_num octets', ...], octets). If *which* is set, it is the message to list.

`POP3.retr(which)`

Retrieve whole message number *which*, and set its seen flag. Result is in form (response, ['line', ...], octets).

`POP3.delete(which)`

Flag message number *which* for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

`POP3.rset()`

Remove any deletion marks for the mailbox.

`POP3.noop()`

Do nothing. Might be used as a keep-alive.

`POP3.quit()`

Signoff : commit changes, unlock mailbox, drop connection.

`POP3.top(which, howmuch)`

Retrieves the message header plus *howmuch* lines of the message after the header of message number *which*. Result is in form (response, ['line', ...], octets).

The POP3 TOP command this method uses, unlike the RETR command, doesn't set the message's seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

`POP3.uidl(which=None)`

Return message digest (unique id) list. If *which* is specified, result contains the unique id for that message in the form 'response mesgnum uid, otherwise result is list (response, ['mesgnum uid', ...], octets).

`POP3.utf8()`

Try to switch to UTF-8 mode. Returns the server response if successful, raises `error_proto` if not. Specified in [RFC 6856](#).

Nouveau dans la version 3.5.

`POP3.stls(context=None)`

Start a TLS session on the active connection as specified in [RFC 2595](#). This is only allowed before user authentication

context parameter is a `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read [Security considerations](#) for best practices.

This method supports hostname checking via `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

Nouveau dans la version 3.4.

Instances of `POP3_SSL` have no additional methods. The interface of this subclass is identical to its parent.

21.14.2 POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages :

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

At the end of the module, there is a test section that contains a more extensive example of usage.

21.15 imaplib — IMAP4 protocol client

Code source : [Lib/imaplib.py](#)

This module defines three classes, *IMAP4*, *IMAP4_SSL* and *IMAP4_stream*, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in [RFC 2060](#). It is backward compatible with IMAP4 ([RFC 1730](#)) servers, but note that the STATUS command is not supported in IMAP4.

Three classes are provided by the *imaplib* module, *IMAP4* is the base class :

class *imaplib.IMAP4* (*host=""*, *port=IMAP4_PORT*)

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If *host* is not specified, '' (the local host) is used. If *port* is omitted, the standard IMAP4 port (143) is used.

The *IMAP4* class supports the *with* statement. When used like this, the IMAP4 LOGOUT command is issued automatically when the *with* statement exits. E.g. :

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

Modifié dans la version 3.5 : La prise en charge de l'instruction *with* a été ajoutée.

Three exceptions are defined as attributes of the *IMAP4* class :

exception *IMAP4.error*

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

exception *IMAP4.abort*

IMAP4 server errors cause this exception to be raised. This is a sub-class of *IMAP4.error*. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

exception *IMAP4.readonly*

This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of *IMAP4.error*. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections :


```
class imaplib.IMAP4_SSL (host="", port=IMAP4_SSL_PORT, keyfile=None, certfile=None,
                        ssl_context=None)
```

This is a subclass derived from *IMAP4* that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If *host* is not specified, `' '` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *ssl_context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

keyfile and *certfile* are a legacy alternative to *ssl_context* - they can point to PEM-formatted private key and certificate chain files for the SSL connection. Note that the *keyfile/certfile* parameters are mutually exclusive with *ssl_context*, a *ValueError* is raised if *keyfile/certfile* is provided along with *ssl_context*.

Modifié dans la version 3.3 : *ssl_context* parameter added.

Modifié dans la version 3.4 : The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Obsolète depuis la version 3.6 : *keyfile* and *certfile* are deprecated in favor of *ssl_context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

The second subclass allows for connections created by a child process :

```
class imaplib.IMAP4_stream (command)
```

This is a subclass derived from *IMAP4* that connects to the `stdin/stdout` file descriptors created by passing *command* to `subprocess.Popen()`.

The following utility functions are defined :

```
imaplib.Internaldate2tuple (datestr)
```

Parse an IMAP4 INTERNALDATE string and return corresponding local time. The return value is a *time.struct_time* tuple or None if the string has wrong format.

```
imaplib.Int2AP (num)
```

Converts an integer into a string representation using characters from the set [A .. P].

```
imaplib.ParseFlags (flagstr)
```

Converts an IMAP4 FLAGS response to a tuple of individual flags.

```
imaplib.Time2Internaldate (date_time)
```

Convert *date_time* to an IMAP4 INTERNALDATE representation. The return value is a string in the form : `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by *time.time()*), a 9-tuple representing local time an instance of *time.struct_time* (as returned by *time.localtime()*), an aware instance of *datetime.datetime*, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes ; in particular, after an EXPUNGE command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

Voir aussi :

Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington's *IMAP Information Center* (<https://www.washington.edu/imap/>).

21.15.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for `AUTHENTICATE`, and the last argument to `APPEND` which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the *password* argument to the `LOGIN` command is always quoted. If you want to avoid having an argument string quoted (eg : the *flags* argument to `STORE`) then enclose the string in parentheses (eg : `r'(\Deleted)'`).

Each command returns a tuple : `(type, [data, ...])` where *type* is usually `'OK'` or `'NO'`, and *data* is either the text from the command response, or mandated results from the command. Each *data* is either a string, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie : “literal” value).

The *message_set* options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number (`'1'`), a range of message numbers (`'2:4'`), or a group of non-contiguous ranges separated by commas (`'1:3,6:9'`). A range can contain an asterisk to indicate an infinite upper bound (`'3:*`').

An *IMAP4* instance has the following methods :

`IMAP4.append(mailbox, flags, date_time, message)`

Append *message* to named mailbox.

`IMAP4.authenticate(mechanism, authobject)`

Authenticate command — requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable *capabilities* in the form `AUTH=mechanism`.

authobject must be a callable object :

```
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be `bytes`. It should return `bytes` *data* that will be base64 encoded and sent to the server. It should return `None` if the client abort response `*` should be sent instead.

Modifié dans la version 3.5 : string usernames and passwords are now encoded to `utf-8` instead of being limited to ASCII.

`IMAP4.check()`

Checkpoint mailbox on server.

`IMAP4.close()`

Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before `LOGOUT`.

`IMAP4.copy(message_set, new_mailbox)`

Copy *message_set* messages onto end of *new_mailbox*.

`IMAP4.create(mailbox)`

Create new mailbox named *mailbox*.

`IMAP4.delete(mailbox)`

Delete old mailbox named *mailbox*.

`IMAP4.deleteacl(mailbox, who)`

Delete the ACLs (remove any rights) set for *who* on mailbox.

`IMAP4.enable(capability)`

Enable *capability* (see [RFC 5161](#)). Most capabilities do not need to be enabled. Currently only the `UTF8=ACCEPT` capability is supported (see [RFC 6855](#)).

Nouveau dans la version 3.5 : The *enable()* method itself, and [RFC 6855](#) support.

- IMAP4.expunge()**
Permanently remove deleted items from selected mailbox. Generates an EXPUNGE response for each deleted message. Returned data contains a list of EXPUNGE message numbers in order received.
- IMAP4.fetch()** (*message_set*, *message_parts*)
Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: " (UID BODY[TEXT]) ". Returned data are tuples of message part envelope and data.
- IMAP4.getacl()** (*mailbox*)
Get the ACLs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.
- IMAP4.getannotation()** (*mailbox*, *entry*, *attribute*)
Retrieve the specified ANNOTATIONS for *mailbox*. The method is non-standard, but is supported by the Cyrus server.
- IMAP4.getquota()** (*root*)
Get the quota *root*'s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.
- IMAP4.getquotaroot()** (*mailbox*)
Get the list of quota roots for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.
- IMAP4.list()** ([*directory*], [*pattern*])
List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of LIST responses.
- IMAP4.login()** (*user*, *password*)
Identify the client using a plaintext password. The *password* will be quoted.
- IMAP4.login_cram_md5()** (*user*, *password*)
Force use of CRAM-MD5 authentication when identifying the client to protect the password. Will only work if the server CAPABILITY response includes the phrase AUTH=CRAM-MD5.
- IMAP4.logout()**
Shutdown connection to server. Returns server BYE response.
- IMAP4.lsub()** (*directory*=*""*, *pattern*=*'*'*)
List subscribed mailbox names in *directory* matching *pattern*. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.
- IMAP4.myrights()** (*mailbox*)
Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).
- IMAP4.namespace()**
Returns IMAP namespaces as defined in [RFC 2342](#).
- IMAP4.noop()**
Send NOOP to server.
- IMAP4.open()** (*host*, *port*)
Opens socket to *port* at *host*. This method is implicitly called by the *IMAP4* constructor. The connection objects established by this method will be used in the *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()*, and *IMAP4.shutdown()* methods. You may override this method.
- IMAP4.partial()** (*message_num*, *message_part*, *start*, *length*)
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.
- IMAP4.proxyauth()** (*user*)
Assume authentication as *user*. Allows an authorised administrator to proxy into any user's mailbox.

`IMAP4.read(size)`

Reads *size* bytes from the remote server. You may override this method.

`IMAP4.readline()`

Reads one line from the remote server. You may override this method.

`IMAP4.recent()`

Prompt server for an update. Returned data is `None` if no new messages, else value of RECENT response.

`IMAP4.rename(oldmailbox, newmailbox)`

Rename mailbox named *oldmailbox* to *newmailbox*.

`IMAP4.response(code)`

Return data for response *code* if received, or `None`. Returns the given code, instead of the usual type.

`IMAP4.search(charset, criterion[, ...])`

Search mailbox for matching messages. *charset* may be `None`, in which case no CHARSET will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error. *charset* must be `None` if the UTF8=ACCEPT capability was enabled using the `enable()` command.

Example :

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

`IMAP4.select(mailbox='INBOX', readonly=False)`

Select a mailbox. Returned data is the count of messages in *mailbox* (EXISTS response). The default *mailbox* is 'INBOX'. If the *readonly* flag is set, modifications to the mailbox are not allowed.

`IMAP4.send(data)`

Sends data to the remote server. You may override this method.

`IMAP4.setacl(mailbox, who, what)`

Set an ACL for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

`IMAP4.setannotation(mailbox, entry, attribute[, ...])`

Set ANNOTATIONS for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

`IMAP4.setquota(root, limits)`

Set the quota *root*'s resource *limits*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

`IMAP4.shutdown()`

Close connection established in open. This method is implicitly called by `IMAP4.logout()`. You may override this method.

`IMAP4.socket()`

Returns socket instance used to connect to server.

`IMAP4.sort(sort_criteria, charset, search_criterion[, ...])`

The sort command is a variant of search with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the *search_criterion* argument(s); a parenthesized list of *sort_criteria*, and the searching *charset*. Note that unlike search, the searching *charset* argument is mandatory. There is also a uid sort command which corresponds to sort the way that uid search corresponds to search. The sort command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an IMAP4rev1 extension command.

IMAP4.**starttls** (*ssl_context=None*)

Send a STARTTLS command. The *ssl_context* argument is optional and should be a `ssl.SSLContext` object. This will enable encryption on the IMAP connection. Please read [Security considerations](#) for best practices.

Nouveau dans la version 3.2.

Modifié dans la version 3.4 : The method now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

IMAP4.**status** (*mailbox, names*)

Request named status conditions for *mailbox*.

IMAP4.**store** (*message_set, command, flag_list*)

Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of [RFC 2060](#) as being one of « **FLAGS** », « **+FLAGS** », or « **-FLAGS** », optionally with a suffix of « **.SILENT** ».

For example, to set the delete flag on all messages :

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

Note : Creating flags containing “]” (for example : « [test] ») violates [RFC 3501](#) (the IMAP protocol). However, `imaplib` has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, `imaplib` nonetheless continues to allow such tags to be created for backward compatibility reasons, and as of python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

IMAP4.**subscribe** (*mailbox*)

Subscribe to new mailbox.

IMAP4.**thread** (*threading_algorithm, charset, search_criterion[, ...]*)

The `thread` command is a variant of `search` with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s) ; a *threading_algorithm*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid thread` command which corresponds to `thread` the way that `uid search` corresponds to `search`. The `thread` command first searches the mailbox for messages that match the given searching criteria using the *charset* argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an IMAP4rev1 extension command.

IMAP4.**uid** (*command, arg[, ...]*)

Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied ; if none are provided, the server will return an error and an exception will be raised.

IMAP4.**unsubscribe** (*mailbox*)

Unsubscribe from old mailbox.

IMAP4.**xatom** (*name[, ...]*)

Allow simple extension commands notified by server in `CAPABILITY` response.

The following attributes are defined on instances of `IMAP4` :

IMAP4.PROTOCOL_VERSION

The most recent supported protocol in the CAPABILITY response from the server.

IMAP4.debug

Integer value to control debugging output. The initialize value is taken from the module variable `Debug`. Values greater than three trace each command.

IMAP4.utf8_enabled

Boolean value that is normally `False`, but is set to `True` if an `enable()` command is successfully issued for the UTF8=ACCEPT capability.

Nouveau dans la version 3.5.

21.15.2 IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages :

```
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

21.16 nntplib — NNTP protocol client

Code source : [Lib/nntplib.py](#)

This module defines the class `NNTP` which implements the client side of the Network News Transfer Protocol. It can be used to implement a news reader or poster, or automated news processors. It is compatible with **RFC 3977** as well as the older **RFC 977** and **RFC 2980**.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles :

```
>>> s = nntplib>NNTP('news.gmane.org')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print('Group', name, 'has', count, 'articles, range', first, 'to', last)
Group gmane.comp.python.committers has 1096 articles, range 1 to 1096
>>> resp, overviews = s.over((last - 9, last))
>>> for id, over in overviews:
...     print(id, nntplib.decode_header(over['subject']))
...
1087 Re: Commit privileges for Łukasz Langa
1088 Re: 3.2 alpha 2 freeze
1089 Re: 3.2 alpha 2 freeze
1090 Re: Commit privileges for Łukasz Langa
1091 Re: Commit privileges for Łukasz Langa
1092 Updated ssh key
```

(suite sur la page suivante)

(suite de la page précédente)

```

1093 Re: Updated ssh key
1094 Re: Updated ssh key
1095 Hello fellow committers!
1096 Re: Hello fellow committers!
>>> s.quit()
'205 Bye!'

```

To post an article from a binary file (this assumes that the article has valid headers, and that you have right to post on the particular newsgroup) :

```

>>> s = nntplib.NNTP('news.gmane.org')
>>> f = open('article.txt', 'rb')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'

```

The module itself defines the following classes :

class `nntplib.NNTP` (*host*, *port*=119, *user*=None, *password*=None, *readermode*=None, *usenetr*=False[, *timeout*])

Return a new `NNTP` object, representing a connection to the NNTP server running on host *host*, listening at port *port*. An optional *timeout* can be specified for the socket connection. If the optional *user* and *password* are provided, or if suitable credentials are present in `.netrc` and the optional flag *usenetr* is true, the `AUTHINFO USER` and `AUTHINFO PASS` commands are used to identify and authenticate the user to the server. If the optional flag *readermode* is true, then a `mode reader` command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as `group`. If you get unexpected `NNTPPermanentErrors`, you might need to set *readermode*. The `NNTP` class supports the `with` statement to unconditionally consume `OSError` exceptions and to close the NNTP connection when done, e.g. :

```

>>> from nntplib import NNTP
>>> with NNTP('news.gmane.org') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.
python.committers')
>>>

```

Modifié dans la version 3.2 : *usenetr* is now `False` by default.

Modifié dans la version 3.3 : La prise en charge de l'instruction `with` a été ajoutée.

class `nntplib.NNTP_SSL` (*host*, *port*=563, *user*=None, *password*=None, *ssl_context*=None, *readermode*=None, *usenetr*=False[, *timeout*])

Return a new `NNTP_SSL` object, representing an encrypted connection to the NNTP server running on host *host*, listening at port *port*. `NNTP_SSL` objects have the same methods as `NNTP` objects. If *port* is omitted, port 563 (NNTPS) is used. *ssl_context* is also optional, and is a `SSLContext` object. Please read [Security considerations](#) for best practices. All other parameters behave the same as for `NNTP`.

Note that SSL-on-563 is discouraged per [RFC 4642](#), in favor of STARTTLS as described below. However, some servers only support the former.

Nouveau dans la version 3.2.

Modifié dans la version 3.4 : The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

exception `nntplib.NNTPError`

Derived from the standard exception `Exception`, this is the base class for all exceptions raised by the `nntplib` module. Instances of this class have the following attribute :

response

The response of the server if available, as a *str* object.

exception `nntplib.NNTPReplyError`

Exception raised when an unexpected reply is received from the server.

exception `nntplib.NNTPTemporaryError`

Exception raised when a response code in the range 400–499 is received.

exception `nntplib.NNTPPermanentError`

Exception raised when a response code in the range 500–599 is received.

exception `nntplib.NNTPProtocolError`

Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5.

exception `nntplib.NNTPDataError`

Exception raised when there is some error in the response data.

21.16.1 NNTP Objects

When connected, *NNTP* and *NNTP_SSL* objects support the following methods and attributes.

Attributes

NNTP.nttp_version

An integer representing the version of the NNTP protocol supported by the server. In practice, this should be 2 for servers advertising **RFC 3977** compliance and 1 for others.

Nouveau dans la version 3.2.

NNTP.nttp_implementation

A string describing the software name and version of the NNTP server, or *None* if not advertised by the server.

Nouveau dans la version 3.2.

Méthodes

The *response* that is returned as the first item in the return tuple of almost all methods is the server's response : a string beginning with a three-digit code. If the server's response indicates an error, the method raises one of the above exceptions.

Many of the following methods take an optional keyword-only argument *file*. When the *file* argument is supplied, it must be either a *file object* opened for binary writing, or the name of an on-disk file to be written to. The method will then write any data returned by the server (except for the response line and the terminating dot) to the file ; any list of lines, tuples or objects that the method normally returns will be empty.

Modifié dans la version 3.2 : Many of the following methods have been reworked and fixed, which makes them incompatible with their 3.1 counterparts.

NNTP.quit()

Send a QUIT command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

NNTP.getwelcome()

Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

NNTP.getcapabilities()

Return the [RFC 3977](#) capabilities advertised by the server, as a *dict* instance mapping capability names to (possibly empty) lists of values. On legacy servers which don't understand the CAPABILITIES command, an empty dictionary is returned instead.

```
>>> s = NNTP('news.gmane.org')
>>> 'POST' in s.getcapabilities()
True
```

Nouveau dans la version 3.2.

NNTP.login(user=None, password=None, usenetrc=True)

Send AUTHINFO commands with the user name and password. If *user* and *password* are None and *usenetrc* is true, credentials from `~/.netrc` will be used if possible.

Unless intentionally delayed, login is normally performed during the *NNTP* object initialization and separately calling this function is unnecessary. To force authentication to be delayed, you must not set *user* or *password* when creating the object, and must set *usenetrc* to False.

Nouveau dans la version 3.2.

NNTP.starttls(context=None)

Send a STARTTLS command. This will enable encryption on the NNTP connection. The *context* argument is optional and should be a *ssl.SSLContext* object. Please read [Security considerations](#) for best practices.

Note that this may not be done after authentication information has been transmitted, and authentication occurs by default if possible during a *NNTP* object initialization. See *NNTP.login()* for information on suppressing this behavior.

Nouveau dans la version 3.2.

Modifié dans la version 3.4 : The method now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

NNTP.newgroups(date, *, file=None)

Send a NEWGROUPS command. The *date* argument should be a *datetime.date* or *datetime.datetime* object. Return a pair (response, groups) where *groups* is a list representing the groups that are new since the given *date*. If *file* is supplied, though, then *groups* will be empty.

```
>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today() - timedelta(days=3))
>>> len(groups)
85
>>> groups[0]
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', flag='m')
```

NNTP.newnews(group, date, *, file=None)

Send a NEWNEWS command. Here, *group* is a group name or '*', and *date* has the same meaning as for *newgroups()*. Return a pair (response, articles) where *articles* is a list of message ids.

This command is frequently disabled by NNTP server administrators.

NNTP.list(group_pattern=None, *, file=None)

Send a LIST or LIST ACTIVE command. Return a pair (response, list) where *list* is a list of tuples representing all the groups available from this NNTP server, optionally matching the pattern string *group_pattern*. Each tuple has the form (group, last, first, flag), where *group* is a group name, *last* and *first* are the last and first article numbers, and *flag* usually takes one of these values :

- y : Local postings and articles from peers are allowed.
- m : The group is moderated and all postings must be approved.
- n : No local postings are allowed, only articles from peers.
- j : Articles from peers are filed in the junk group instead.
- x : No local postings, and articles from peers are ignored.
- =foo.bar : Articles are filed in the foo.bar group instead.

If *flag* has another value, then the status of the newsgroup should be considered unknown.

This command can return very large results, especially if *group_pattern* is not specified. It is best to cache the results offline unless you really need to refresh them.

Modifié dans la version 3.2 : *group_pattern* was added.

NNTP **.descriptions** (*grouppattern*)

Send a LIST NEWSGROUPS command, where *grouppattern* is a wildmat string as specified in [RFC 3977](#) (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair (*response*, *descriptions*), where *descriptions* is a dictionary mapping group names to textual descriptions.

```
>>> resp, descs = s.descriptions('gmmane.comp.python.*')
>>> len(descs)
295
>>> descs.popitem()
('gmmane.comp.python.bio.general', 'BioPython discussion list (Moderated)')
```

NNTP **.description** (*group*)

Get a description for a single group *group*. If more than one group matches (if “group” is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use *descriptions()*.

NNTP **.group** (*name*)

Send a GROUP command, where *name* is the group name. The group is selected as the current group, if it exists. Return a tuple (*response*, *count*, *first*, *last*, *name*) where *count* is the (estimated) number of articles in the group, *first* is the first article number in the group, *last* is the last article number in the group, and *name* is the group name.

NNTP **.over** (*message_spec*, *, *file=None*)

Send an OVER command, or an XOVER command on legacy servers. *message_spec* can be either a string representing a message id, or a (*first*, *last*) tuple of numbers indicating a range of articles in the current group, or a (*first*, *None*) tuple indicating a range of articles starting from *first* to the last article in the current group, or *None* to select the current article in the current group.

Return a pair (*response*, *overviews*). *overviews* is a list of (*article_number*, *overview*) tuples, one for each article selected by *message_spec*. Each *overview* is a dictionary with the same number of items, but this number depends on the server. These items are either message headers (the key is then the lower-cased header name) or metadata items (the key is then the metadata name prepended with “:”). The following items are guaranteed to be present by the NNTP specification :

- the subject, from, date, message-id and references headers
- the `:bytes` metadata: the number of bytes in the entire raw article (including headers and body)
- the `:lines` metadata: the number of lines in the article body

The value of each item is either a string, or *None* if not present.

It is advisable to use the *decode_header()* function on header values when they may contain non-ASCII characters :

```
>>> _, _, first, last, _ = s.group('gmmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'message-id', 'subject
↵']
>>> over['from']
'=?UTF-8?B?Ik1hcnRpb2I2LiBMw7Z3aXMi?= <martin@v.loewis.de>'
>>> nntplib.decode_header(over['from'])
"Martin v. Löwis" <martin@v.loewis.de>
```

Nouveau dans la version 3.2.

NNTP.**help** (*, *file*=None)

Send a HELP command. Return a pair (*response*, *list*) where *list* is a list of help strings.

NNTP.**stat** (*message_spec*=None)

Send a STAT command, where *message_spec* is either a message id (enclosed in '<' and '>') or an article number in the current group. If *message_spec* is omitted or *None*, the current article in the current group is considered. Return a triple (*response*, *number*, *id*) where *number* is the article number and *id* is the message id.

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>')
```

NNTP.**next** ()

Send a NEXT command. Return as for *stat* ().

NNTP.**last** ()

Send a LAST command. Return as for *stat* ().

NNTP.**article** (*message_spec*=None, *, *file*=None)

Send an ARTICLE command, where *message_spec* has the same meaning as for *stat* (). Return a tuple (*response*, *info*) where *info* is a *namedtuple* with three attributes *number*, *message_id* and *lines* (in that order). *number* is the article number in the group (or 0 if the information is not available), *message_id* the message id as a string, and *lines* a list of lines (without terminating newlines) comprising the raw message including headers and body.

```
>>> resp, info = s.article('<20030112190404.GE29873@epoch.metaslash.com>')
>>> info.number
0
>>> info.message_id
'<20030112190404.GE29873@epoch.metaslash.com>'
>>> len(info.lines)
65
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
b'From: Neal Norwitz <neal@metaslash.com>'
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b'', b'Neal']
```

NNTP.**head** (*message_spec*=None, *, *file*=None)

Same as *article* (), but sends a HEAD command. The *lines* returned (or written to *file*) will only contain the message headers, not the body.

NNTP.**body** (*message_spec*=None, *, *file*=None)

Same as *article* (), but sends a BODY command. The *lines* returned (or written to *file*) will only contain the message body, not the headers.

NNTP.**post** (*data*)

Post an article using the POST command. The *data* argument is either a *file object* opened for binary reading, or any iterable of bytes objects (representing raw lines of the article to be posted). It should represent a well-formed news article, including the required headers. The *post* () method automatically escapes lines beginning with . and appends the termination line.

If the method succeeds, the server's response is returned. If the server refuses posting, a *NNTPReplyError* is raised.

NNTP.**ihave** (*message_id*, *data*)

Send an IHAVE command. *message_id* is the id of the message to send to the server (enclosed in '<' and '>'). The *data* parameter and the return value are the same as for `post()`.

NNTP.**date** ()

Return a pair (*response*, *date*). *date* is a `datetime` object containing the current date and time of the server.

NNTP.**slave** ()

Send a SLAVE command. Return the server's *response*.

NNTP.**set_debuglevel** (*level*)

Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

The following are optional NNTP extensions defined in [RFC 2980](#). Some of them have been superseded by newer commands in [RFC 3977](#).

NNTP.**xhdr** (*hdr*, *str*, *, *file*=None)

Send an XHDR command. The *hdr* argument is a header keyword, e.g. 'subject'. The *str* argument should have the form 'first-last' where *first* and *last* are the first and last article numbers to search. Return a pair (*response*, *list*), where *list* is a list of pairs (*id*, *text*), where *id* is an article number (as a string) and *text* is the text of the requested header for that article. If the *file* parameter is supplied, then the output of the XHDR command is stored in a file. If *file* is a string, then the method will open a file with that name, write to it then close it. If *file* is a `file object`, then it will start calling `write()` on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

NNTP.**xover** (*start*, *end*, *, *file*=None)

Send an XOVER command. *start* and *end* are article numbers delimiting the range of articles to select. The return value is the same of for `over()`. It is recommended to use `over()` instead, since it will automatically use the newer OVER command if available.

NNTP.**xpath** (*id*)

Return a pair (*resp*, *path*), where *path* is the directory path to the article with message ID *id*. Most of the time, this extension is not enabled by NNTP server administrators.

Obsolète depuis la version 3.3 : The XPATH extension is not actively used.

21.16.2 Fonctions utilitaires

The module also defines the following utility function :

`nntplib.decode_header` (*header_str*)

Decode a header value, un-escaping any escaped non-ASCII characters. *header_str* must be a `str` object. The unescaped value is returned. Using this function is recommended to display some headers in a human readable form :

```
>>> decode_header("Some subject")
'Some subject'
>>> decode_header("=?ISO-8859-15?Q?D=E9buter_en_Python?=")
'Débuter en Python'
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```

21.17 smtplib — SMTP protocol client

Code source : [Lib/smtplib.py](#)

The `smtplib` module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult [RFC 821](#) (Simple Mail Transfer Protocol) and [RFC 1869](#) (SMTP Service Extensions).

class `smtplib.SMTP` (*host*="", *port*=0, *local_hostname*=None[, *timeout*], *source_address*=None)

An `SMTP` instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional *host* and *port* parameters are given, the `SMTP.connect()` method is called with those parameters during initialization. If specified, *local_hostname* is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using `socket.getfqdn()`. If the `connect()` call returns anything other than a success code, an `SMTPConnectError` is raised. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, `socket.timeout` is raised. The optional *source_address* parameter allows binding to some specific source address in a machine with multiple network interfaces, and/or to some specific source TCP port. It takes a 2-tuple (*host*, *port*), for the socket to bind to as its source address before connecting. If omitted (or if *host* or *port* are '' and/or 0 respectively) the OS default behavior will be used.

For normal use, you should only require the initialization/connect, `sendmail()`, and `SMTP.quit()` methods. An example is included below.

The `SMTP` class supports the `with` statement. When used like this, the SMTP QUIT command is issued automatically when the `with` statement exits. E.g. :

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

Modifié dans la version 3.3 : La prise en charge de l'instruction `with` a été ajoutée.

Modifié dans la version 3.3 : *source_address* argument was added.

Nouveau dans la version 3.5 : The SMTPUTF8 extension ([RFC 6531](#)) is now supported.

class `smtplib.SMTP_SSL` (*host*="", *port*=0, *local_hostname*=None, *keyfile*=None, *certfile*=None[, *timeout*], *context*=None, *source_address*=None)

An `SMTP_SSL` instance behaves exactly the same as instances of `SMTP`. `SMTP_SSL` should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If *host* is not specified, the local host is used. If *port* is zero, the standard SMTP-over-SSL port (465) is used. The optional arguments *local_hostname*, *timeout* and *source_address* have the same meaning as they do in the `SMTP` class. *context*, also optional, can contain a `SSLContext` and allows configuring various aspects of the secure connection. Please read [Security considerations](#) for best practices.

keyfile and *certfile* are a legacy alternative to *context*, and can point to a PEM formatted private key and certificate chain file for the SSL connection.

Modifié dans la version 3.3 : *context* was added.

Modifié dans la version 3.3 : *source_address* argument was added.

Modifié dans la version 3.4 : The class now supports hostname check with `ssl.SSLContext.check_hostname` and Server Name Indication (see `ssl.HAS_SNI`).

Obsolète depuis la version 3.6 : *keyfile* and *certfile* are deprecated in favor of *context*. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system's trusted CA certificates for you.

class `smtplib.LMTP` (*host=""*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*)

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular `host:port` server. The optional arguments `local_hostname` and `source_address` have the same meaning as they do in the [SMTP](#) class. To specify a Unix socket, you must use an absolute path for *host*, starting with a `/`.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don't support or require any authentication, but your mileage might vary.

A nice selection of exceptions is defined as well :

exception `smtplib.SMTPException`

Subclass of [OSError](#) that is the base exception class for all the other exceptions provided by this module.

Modifié dans la version 3.4 : `SMTPException` became subclass of [OSError](#)

exception `smtplib.SMTPServerDisconnected`

This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the [SMTP](#) instance before connecting it to a server.

exception `smtplib.SMTPResponseException`

Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

exception `smtplib.SMTPSenderRefused`

Sender address refused. In addition to the attributes set by on all [SMTPResponseException](#) exceptions, this sets "sender" to the string that the SMTP server refused.

exception `smtplib.SMTPRecipientsRefused`

All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary of exactly the same sort as [SMTP.sendmail\(\)](#) returns.

exception `smtplib.SMTPDataError`

The SMTP server refused to accept the message data.

exception `smtplib.SMTPConnectError`

Error occurred during establishment of a connection with the server.

exception `smtplib.SMTPHeloError`

The server refused our HELO message.

exception `smtplib.SMTPNotSupportedError`

The command or option attempted is not supported by the server.

Nouveau dans la version 3.5.

exception `smtplib.SMTPAuthenticationError`

SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.

Voir aussi :

RFC 821 - Simple Mail Transfer Protocol Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

21.17.1 SMTP Objects

An *SMTP* instance has the following methods :

SMTP.set_debuglevel (*level*)

Set the debug output level. A value of 1 or `True` for *level* results in debug messages for connection and for all messages sent to and received from the server. A value of 2 for *level* results in these messages being timestamped.

Modifié dans la version 3.5 : Added debuglevel 2.

SMTP.doccmd (*cmd*, *args=""*)

Send a command *cmd* to the server. The optional argument *args* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, *SMTPServerDisconnected* will be raised.

SMTP.connect (*host='localhost'*, *port=0*)

Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (':') followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

SMTP.helo (*name=""*)

Identify yourself to the SMTP server using HELO. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the *helo_resp* attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the *sendmail()* when necessary.

SMTP.ehlo (*name=""*)

Identify yourself to an ESMTP server using EHLO. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by *has_extn()*. Also sets several informational attributes : the message returned by the server is stored as the *ehlo_resp* attribute, *does_esmtp* is set to true or false depending on whether the server supports ESMTP, and *esmtp_features* will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use *has_extn()* before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by *sendmail()* when necessary.

SMTP.ehlo_or_helo_if_needed ()

This method calls *ehlo()* and/or *helo()* if there has been no previous EHLO or HELO command this session. It tries ESMTP EHLO first.

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTP.has_extn (*name*)

Return *True* if *name* is in the set of SMTP service extensions returned by the server, *False* otherwise. Case is ignored.

SMTP.verify (*address*)

Check the validity of an address on this server using SMTP VRFY. Returns a tuple consisting of code 250 and a full **RFC 822** address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

Note : Many sites disable SMTP VRFY in order to foil spammers.

SMTP.**login** (*user*, *password*, *, *initial_response_ok*=True)

Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions :

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPAuthenticationError The server didn't accept the username/password combination.

SMTPNotSupportedError The AUTH command is not supported by the server.

SMTPException No suitable authentication method was found.

Each of the authentication methods supported by *smtplib* are tried in turn if they are advertised as supported by the server. See *auth()* for a list of supported authentication methods. *initial_response_ok* is passed through to *auth()*.

Optional keyword argument *initial_response_ok* specifies whether, for authentication methods that support it, an « initial response » as specified in **RFC 4954** can be sent along with the AUTH command, rather than requiring a challenge/response.

Modifié dans la version 3.5 : **SMTPNotSupportedError** may be raised, and the *initial_response_ok* parameter was added.

SMTP.**auth** (*mechanism*, *authobject*, *, *initial_response_ok*=True)

Issue an SMTP AUTH command for the specified authentication *mechanism*, and handle the challenge response via *authobject*.

mechanism specifies which authentication mechanism is to be used as argument to the AUTH command ; the valid values are those listed in the *auth* element of *esmtplib.features*.

authobject must be a callable object taking an optional single argument :

data = authobject(challenge=None)

If optional keyword argument *initial_response_ok* is true, *authobject()* will be called first with no argument. It can return the **RFC 4954** « initial response » ASCII str which will be encoded and sent with the AUTH command as below. If the *authobject()* does not support an initial response (e.g. because it requires a challenge), it should return None when called with *challenge=None*. If *initial_response_ok* is false, then *authobject()* will not be called first with None.

If the initial response check returns None, or if *initial_response_ok* is false, *authobject()* will be called to process the server's challenge response ; the *challenge* argument it is passed will be a bytes. It should return ASCII str *data* that will be base64 encoded and sent to the server.

The SMTP class provides *authobjects* for the CRAM-MD5, PLAIN, and LOGIN mechanisms ; they are named SMTP.*auth_cram_md5*, SMTP.*auth_plain*, and SMTP.*auth_login* respectively. They all require that the *user* and *password* properties of the SMTP instance are set to appropriate values.

User code does not normally need to call *auth* directly, but can instead call the *login()* method, which will try each of the above mechanisms in turn, in the order listed. *auth* is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by *smtplib*.

Nouveau dans la version 3.5.

SMTP.**starttls** (*keyfile*=None, *certfile*=None, *context*=None)

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call *ehlo()* again.

If *keyfile* and *certfile* are provided, they are used to create an *ssl.SSLContext*.

Optional *context* parameter is an *ssl.SSLContext* object ; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be None.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first.

Obsolète depuis la version 3.6 : *keyfile* and *certfile* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

SMTPHelloError The server didn't reply properly to the HELO greeting.

SMTPNotSupportedError The server does not support the STARTTLS extension.

RuntimeError SSL/TLS support is not available to your Python interpreter.

Modifié dans la version 3.3 : *context* was added.

Modifié dans la version 3.4 : The method now supports hostname check with `SSLContext.check_hostname` and *Server Name Indicator* (see [HAS_SNI](#)).

Modifié dans la version 3.5 : The error raised for lack of STARTTLS support is now the **SMTPNotSupportedError** subclass instead of the base **SMTPException**.

SMTP **.sendmail** (*from_addr*, *to_addrs*, *msg*, *mail_options*=(), *rcpt_options*=())

Send mail. The required arguments are an [RFC 822](#) from-address string, a list of [RFC 822](#) to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as `8bitmime`) to be used in MAIL FROM commands as *mail_options*. ESMTP options (such as DSN commands) that should be used with all RCPT commands can be passed as *rcpt_options*. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as `mail()`, `rcpt()` and `data()` to send the message.)

Note : The *from_addr* and *to_addrs* parameters are used to construct the message envelope used by the transport agents. `sendmail` does not modify the message headers in any way.

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the `ascii` codec, and lone `\r` and `\n` characters are converted to `\r\n` characters. A byte string is not modified.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If EHLO fails, HELO will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

If `SMTPUTF8` is included in *mail_options*, and the server supports it, *from_addr* and *to_addrs* may contain non-ASCII characters.

This method may raise the following exceptions :

SMTPRecipientsRefused All recipients were refused. Nobody got the mail. The *recipients* attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

SMTPHelloError The server didn't reply properly to the HELO greeting.

SMTPSenderRefused The server didn't accept the *from_addr*.

SMTPDataError The server replied with an unexpected error code (other than a refusal of a recipient).

SMTPNotSupportedError `SMTPUTF8` was given in the *mail_options* but is not supported by the server.

Unless otherwise noted, the connection will be open even after an exception is raised.

Modifié dans la version 3.2 : *msg* may be a byte string.

Modifié dans la version 3.5 : `SMTPUTF8` support added, and **SMTPNotSupportedError** may be raised if `SMTPUTF8` is specified but the server does not support it.

SMTP **.send_message** (*msg*, *from_addr*=None, *to_addrs*=None, *mail_options*=(), *rcpt_options*=())

This is a convenience method for calling `sendmail()` with the message represented by an `email.message.Message` object. The arguments have the same meaning as for `sendmail()`, except that *msg* is a `Message` object.

If *from_addr* is None or *to_addrs* is None, `send_message` fills those arguments with addresses extracted from the headers of *msg* as specified in [RFC 5322](#) : *from_addr* is set to the *Sender* field if it is present, and otherwise to the *From* field. *to_addrs* combines the values (if any) of the *To*, *Cc*, and *Bcc* fields from *msg*. If exactly one

set of *Resent-** headers appear in the message, the regular headers are ignored and the *Resent-** headers are used instead. If the message contains more than one set of *Resent-** headers, a *ValueError* is raised, since there is no way to unambiguously detect the most recent set of *Resent-* headers.

`send_message` serializes *msg* using *BytesGenerator* with `\r\n` as the *linesep*, and calls `sendmail()` to transmit the resulting message. Regardless of the values of *from_addr* and *to_addrs*, `send_message` does not transmit any *Bcc* or *Resent-Bcc* headers that may appear in *msg*. If any of the addresses in *from_addr* and *to_addrs* contain non-ASCII characters and the server does not advertise SMTPUTF8 support, an *SMTPNotSupported* error is raised. Otherwise the Message is serialized with a clone of its *policy* with the *utf8* attribute set to True, and SMTPUTF8 and BODY=8BITMIME are added to *mail_options*.

Nouveau dans la version 3.2.

Nouveau dans la version 3.5 : Support for internationalized addresses (SMTPUTF8).

`SMTP.quit()`

Terminate the SMTP session and close the connection. Return the result of the SMTP QUIT command.

Low-level methods corresponding to the standard SMTP/ESMTP commands HELP, RSET, NOOP, MAIL, RCPT, and DATA are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

21.17.2 SMTP Example

This example prompts the user for addresses needed in the message envelope (“To” and “From” addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn’t do any processing of the **RFC 822** headers. In particular, the “To” and “From” addresses must be included in the message headers explicitly.

```
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
      % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Note : In general, you will want to use the *email* package’s features to construct an email message, which you can then

send via `send_message()`; see *email: Examples*.

21.18 smtpd — SMTP Server

Code source : [Lib/smtpd.py](#)

This module offers several classes to implement SMTP (email) servers.

Voir aussi :

The `aiosmtpd` package is a recommended replacement for this module. It is based on `asyncio` and provides a more straightforward API. `smtpd` should be considered deprecated.

Several server implementations are present; one is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies.

Additionally the SMTPChannel may be extended to implement very specific interaction behaviour with SMTP clients.

The code supports [RFC 5321](#), plus the [RFC 1870](#) SIZE and [RFC 6531](#) SMTPUTF8 extensions.

21.18.1 SMTPServer Objects

class `smtpd.SMTPServer`(*localaddr*, *remoteaddr*, *data_size_limit*=33554432, *map*=None, *enable_SMTPUTF8*=False, *decode_data*=False)

Create a new `SMTPServer` object, which binds to local address *localaddr*. It will treat *remoteaddr* as an upstream SMTP relayer. Both *localaddr* and *remoteaddr* should be a (*host*, *port*) tuple. The object inherits from `asyncore.dispatcher`, and so will insert itself into `asyncore`'s event loop on instantiation.

data_size_limit specifies the maximum number of bytes that will be accepted in a DATA command. A value of None or 0 means no limit.

map is the socket map to use for connections (an initially empty dictionary is a suitable value). If not specified the `asyncore` global socket map is used.

enable_SMTPUTF8 determines whether the SMTPUTF8 extension (as defined in [RFC 6531](#)) should be enabled. The default is False. When True, SMTPUTF8 is accepted as a parameter to the MAIL command and when present is passed to `process_message()` in the `kwargs['mail_options']` list. *decode_data* and *enable_SMTPUTF8* cannot be set to True at the same time.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. When *decode_data* is False (the default), the server advertises the 8BITMIME extension ([RFC 6152](#)), accepts the BODY=8BITMIME parameter to the MAIL command, and when present passes it to `process_message()` in the `kwargs['mail_options']` list. *decode_data* and *enable_SMTPUTF8* cannot be set to True at the same time.

process_message(*peer*, *mailfrom*, *rcpttos*, *data*, ***kwargs*)

Raise a `NotImplementedError` exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as *remoteaddr* will be available as the `_remoteaddr` attribute. *peer* is the remote host's address, *mailfrom* is the envelope originator, *rcpttos* are the envelope recipients and *data* is a string containing the contents of the e-mail (which should be in [RFC 5321](#) format).

If the *decode_data* constructor keyword is set to True, the *data* argument will be a unicode string. If it is set to False, it will be a bytes object.

kwargs is a dictionary containing additional information. It is empty if *decode_data*=True was given as an init argument, otherwise it contains the following keys :

mail_options : a list of all received parameters to the MAIL command (the elements are uppercase strings; example : ['BODY=8BITMIME' , 'SMTPUTF8']).

rcpt_options : same as *mail_options* but for the RCPT command. Currently no RCPT TO options are supported, so for now this will always be an empty list.

Implementations of *process_message* should use the ****kwargs** signature to accept arbitrary keyword arguments, since future feature enhancements may add keys to the kwargs dictionary.

Return *None* to request a normal 250 Ok response; otherwise return the desired response string in **RFC 5321** format.

channel_class

Override this in subclasses to use a custom *SMTPChannel* for managing SMTP clients.

Nouveau dans la version 3.4 : The *map* constructor argument.

Modifié dans la version 3.5 : *localaddr* and *remoteaddr* may now contain IPv6 addresses.

Nouveau dans la version 3.5 : The *decode_data* and *enable_SMTPUTF8* constructor parameters, and the *kwargs* parameter to *process_message()* when *decode_data* is *False*.

Modifié dans la version 3.6 : *decode_data* is now *False* by default.

21.18.2 DebuggingServer Objects

class *smtpd.DebuggingServer* (*localaddr*, *remoteaddr*)

Create a new debugging server. Arguments are as per *SMTPServer*. Messages will be discarded, and printed on stdout.

21.18.3 PureProxy Objects

class *smtpd.PureProxy* (*localaddr*, *remoteaddr*)

Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*. Note that running this has a good chance to make you into an open relay, so please be careful.

21.18.4 MailmanProxy Objects

class *smtpd.MailmanProxy* (*localaddr*, *remoteaddr*)

Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.

21.18.5 SMTPChannel Objects

class *smtpd.SMTPChannel* (*server*, *conn*, *addr*, *data_size_limit*=33554432, *map*=*None*, *enable_SMTPUTF8*=*False*, *decode_data*=*False*)

Create a new *SMTPChannel* object which manages the communication between the server and a single SMTP client.

conn and *addr* are as per the instance variables described below.

data_size_limit specifies the maximum number of bytes that will be accepted in a DATA command. A value of *None* or 0 means no limit.

enable_SMTPUTF8 determines whether the SMTPUTF8 extension (as defined in **RFC 6531**) should be enabled. The default is *False*. *decode_data* and *enable_SMTPUTF8* cannot be set to *True* at the same time.

A dictionary can be specified in *map* to avoid using a global socket map.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. The default is *False*. *decode_data* and *enable_SMTPUTF8* cannot be set to *True* at the same time.

To use a custom SMTPChannel implementation you need to override the *SMTPServer.channel_class* of your *SMTPServer*.

Modifié dans la version 3.5 : The `decode_data` and `enable_SMTPUTF8` parameters were added.

Modifié dans la version 3.6 : `decode_data` is now `False` by default.

The `SMTPChannel` has the following instance variables :

smtp_server

Holds the `SMTPServer` that spawned this channel.

conn

Holds the socket object connecting to the client.

addr

Holds the address of the client, the second value returned by `socket.accept`

received_lines

Holds a list of the line strings (decoded using UTF-8) received from the client. The lines have their `"\r\n"` line ending translated to `"\n"`.

smtp_state

Holds the current state of the channel. This will be either `COMMAND` initially and then `DATA` after the client sends a « DATA » line.

seen_greeting

Holds a string containing the greeting sent by the client in its « HELO ».

mailfrom

Holds a string containing the address identified in the « MAIL FROM : » line from the client.

rcpttos

Holds a list of strings containing the addresses identified in the « RCPT TO : » lines from the client.

received_data

Holds a string containing all of the data sent by the client during the `DATA` state, up to but not including the terminating `"\r\n.\r\n"`.

fqdn

Holds the fully-qualified domain name of the server as returned by `socket.getfqdn()`.

peer

Holds the name of the client peer as returned by `conn.getpeername()` where `conn` is `conn`.

The `SMTPChannel` operates by invoking methods named `smtp_<command>` upon reception of a command line from the client. Built into the base `SMTPChannel` class are methods for handling the following commands (and responding to them appropriately) :

Com-mande	Action taken
HELO	Accepts the greeting from the client and stores it in <code>seen_greeting</code> . Sets server to base command mode.
EHLO	Accepts the greeting from the client and stores it in <code>seen_greeting</code> . Sets server to extended command mode.
NOOP	Takes no action.
QUIT	Closes the connection cleanly.
MAIL	Accepts the « MAIL FROM : » syntax and stores the supplied address as <code>mailfrom</code> . In extended command mode, accepts the RFC 1870 SIZE attribute and responds appropriately based on the value of <code>data_size_limit</code> .
RCPT	Accepts the « RCPT TO : » syntax and stores the supplied addresses in the <code>rcpttos</code> list.
RSET	Resets the <code>mailfrom</code> , <code>rcpttos</code> , and <code>received_data</code> , but not the greeting.
DATA	Sets the internal state to <code>DATA</code> and stores remaining lines from the client in <code>received_data</code> until the terminator <code>"\r\n.\r\n"</code> is received.
HELP	Returns minimal information on command syntax
VERFY	Returns code 252 (the server doesn't know if the address is valid)
EXPN	Reports that the command is not implemented.

21.19 telnetlib — Telnet client

Code source : [Lib/telnetlib.py](#)

The `telnetlib` module provides a `Telnet` class that implements the Telnet protocol. See [RFC 854](#) for details about the protocol. In addition, it provides symbolic constants for the protocol characters (see below), and for the telnet options. The symbolic names of the telnet options follow the definitions in `arpa/telnet.h`, with the leading `TELOPT_` removed. For symbolic names of options which are traditionally not included in `arpa/telnet.h`, see the module source itself.

The symbolic constants for the telnet commands are : IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

class `telnetlib.Telnet` (*host=None, port=0[, timeout]*)

`Telnet` represents a connection to a Telnet server. The instance is initially not connected by default ; the `open()` method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor too, in which case the connection to the server will be established before the constructor returns. The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not reopen an already connected instance.

This class has many `read_*()` methods. Note that some of them raise `EOFError` when the end of the connection is read, because they can return an empty string for other reasons. See the individual descriptions below.

A `Telnet` object is a context manager and can be used in a `with` statement. When the `with` block ends, the `close()` method is called :

```
>>> from telnetlib import Telnet
>>> with Telnet('localhost', 23) as tn:
...     tn.interact()
... 
```

Modifié dans la version 3.6 : Context manager support added

Voir aussi :

RFC 854 - Telnet Protocol Specification Definition of the Telnet protocol.

21.19.1 Telnet Objects

`Telnet` instances have the following methods :

`Telnet.read_until` (*expected, timeout=None*)

Read until a given byte string, *expected*, is encountered or until *timeout* seconds have passed.

When no match is found, return whatever is available instead, possibly empty bytes. Raise `EOFError` if the connection is closed and no cooked data is available.

`Telnet.read_all` ()

Read all data until EOF as bytes ; block until connection closed.

`Telnet.read_some` ()

Read at least one byte of cooked data unless EOF is hit. Return `b''` if EOF is hit. Block if no data is immediately available.

`Telnet.read_very_eager()`
 Read everything that can be without blocking in I/O (eager).
 Raise `EOFError` if connection closed and no cooked data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

`Telnet.read_eager()`
 Read readily available data.
 Raise `EOFError` if connection closed and no cooked data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

`Telnet.read_lazy()`
 Process and return data already in the queues (lazy).
 Raise `EOFError` if connection closed and no data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

`Telnet.read_very_lazy()`
 Return any data available in the cooked queue (very lazy).
 Raise `EOFError` if connection closed and no data available. Return `b''` if no cooked data available otherwise. This method never blocks.

`Telnet.read_sb_data()`
 Return the data collected between a SB/SE pair (suboption begin/end). The callback should access these data when it was invoked with a SE command. This method never blocks.

`Telnet.open(host, port=0[, timeout])`
 Connect to a host. The optional second argument is the port number, which defaults to the standard Telnet port (23). The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).
 Do not try to reopen an already connected instance.

`Telnet.msg(msg, *args)`
 Print a debug message when the debug level is `> 0`. If extra arguments are present, they are substituted in the message using the standard string formatting operator.

`Telnet.set_debuglevel(debuglevel)`
 Set the debug level. The higher the value of `debuglevel`, the more debug output you get (on `sys.stdout`).

`Telnet.close()`
 Ferme la connexion.

`Telnet.get_socket()`
 Return the socket object used internally.

`Telnet.fileno()`
 Return the file descriptor of the socket object used internally.

`Telnet.write(buffer)`
 Write a byte string to the socket, doubling any IAC characters. This can block if the connection is blocked. May raise `OSError` if the connection is closed.
 Modifié dans la version 3.3 : This method used to raise `socket.error`, which is now an alias of `OSError`.

`Telnet.interact()`
 Interaction function, emulates a very dumb Telnet client.

`Telnet.mt_interact()`
 Multithreaded version of `interact()`.

`Telnet.expect(list, timeout=None)`
 Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (*regex objects*) or uncompiled (byte strings). The optional second argument is a timeout, in seconds; the default is to block indefinitely.

Return a tuple of three items : the index in the list of the first regular expression that matches; the match object returned; and the bytes read up till and including the match.

If end of file is found and no bytes were read, raise *EOFError*. Otherwise, when nothing matches, return `(-1, None, data)` where *data* is the bytes received so far (may be empty bytes if a timeout happened).

If a regular expression ends with a greedy match (such as `.*`) or if more than one expression can match the same input, the results are non-deterministic, and may depend on the I/O timing.

`Telnet.set_option_negotiation_callback(callback)`

Each time a telnet option is read on the input flow, this *callback* (if set) is called with the following parameters : `callback(telnet socket, command (DO/DONT/WILL/WONT), option)`. No other action is done afterwards by telnetlib.

21.19.2 Telnet Example

A simple example illustrating typical use :

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

21.20 uuid — UUID objects according to RFC 4122

Source code : [Lib/uuid.py](#)

This module provides immutable *UUID* objects (the *UUID* class) and the functions `uuid1()`, `uuid3()`, `uuid4()`, `uuid5()` for generating version 1, 3, 4, and 5 UUIDs as specified in **RFC 4122**.

If all you want is a unique ID, you should probably call `uuid1()` or `uuid4()`. Note that `uuid1()` may compromise privacy since it creates a UUID containing the computer's network address. `uuid4()` creates a random UUID.

class `uuid.UUID` (*hex=None, bytes=None, bytes_le=None, fields=None, int=None, version=None*)

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes in big-endian order as the *bytes* argument, a string of 16 bytes in little-endian order as the *bytes_le* argument, a tuple of six integers (32-bit *time_low*, 16-bit *time_mid*, 16-bit *time_hi_version*, 8-bit *clock_seq_hi_variant*, 8-bit *clock_seq_low*, 48-bit *node*) as the *fields*

argument, or a single 128-bit integer as the *int* argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID :

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
            b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of *hex*, *bytes*, *bytes_le*, *fields*, or *int* must be given. The *version* argument is optional; if given, the resulting UUID will have its variant and version number set according to [RFC 4122](#), overriding bits in the given *hex*, *bytes*, *bytes_le*, *fields*, or *int*.

Comparison of UUID objects are made by way of comparing their *UUID.int* attributes. Comparison with a non-UUID object raises a *TypeError*.

str(uuid) returns a string in the form 12345678-1234-5678-1234-567812345678 where the 32 hexadecimal digits represent the UUID.

UUID instances have these read-only attributes :

UUID.bytes

The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

UUID.bytes_le

The UUID as a 16-byte string (with *time_low*, *time_mid*, and *time_hi_version* in little-endian byte order).

UUID.fields

A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes :

Field	Signification
<i>time_low</i>	the first 32 bits of the UUID
<i>time_mid</i>	the next 16 bits of the UUID
<i>time_hi_version</i>	the next 16 bits of the UUID
<i>clock_seq_hi_variant</i>	the next 8 bits of the UUID
<i>clock_seq_low</i>	the next 8 bits of the UUID
<i>node</i>	the last 48 bits of the UUID
<i>time</i>	the 60-bit timestamp
<i>clock_seq</i>	the 14-bit sequence number

UUID.hex

The UUID as a 32-character hexadecimal string.

UUID.int

The UUID as a 128-bit integer.

UUID.urn

The UUID as a URN as specified in [RFC 4122](#).

UUID.variant

The UUID variant, which determines the internal layout of the UUID. This will be one of the constants *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT*, or *RESERVED_FUTURE*.

UUID.version

The UUID version number (1 through 5, meaningful only when the variant is *RFC_4122*).

The *uuid* module defines the following functions :

`uuid.getnode()`

Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with its eighth bit set to 1 as recommended in [RFC 4122](#). « Hardware address » means the MAC address of a network interface, and on a machine with multiple network interfaces the MAC address of any one of them may be returned.

`uuid.uuid1 (node=None, clock_seq=None)`

Generate a UUID from a host ID, sequence number, and the current time. If *node* is not given, `getnode()` is used to obtain the hardware address. If *clock_seq* is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.

`uuid.uuid3 (namespace, name)`

Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a string).

`uuid.uuid4 ()`

Generate a random UUID.

`uuid.uuid5 (namespace, name)`

Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a string).

The `uuid` module defines the following namespace identifiers for use with `uuid3()` or `uuid5()`.

`uuid.NAMESPACE_DNS`

When this namespace is specified, the *name* string is a fully-qualified domain name.

`uuid.NAMESPACE_URL`

When this namespace is specified, the *name* string is a URL.

`uuid.NAMESPACE_OID`

When this namespace is specified, the *name* string is an ISO OID.

`uuid.NAMESPACE_X500`

When this namespace is specified, the *name* string is an X.500 DN in DER or a text output format.

The `uuid` module defines the following constants for the possible values of the `variant` attribute :

`uuid.RESERVED_NCS`

Reserved for NCS compatibility.

`uuid.RFC_4122`

Specifies the UUID layout given in [RFC 4122](#).

`uuid.RESERVED_MICROSOFT`

Reserved for Microsoft compatibility.

`uuid.RESERVED_FUTURE`

Reserved for future definition.

Voir aussi :

[RFC 4122](#) - A Universally Unique Identifier (UUID) URN Namespace This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.

21.20.1 Exemple

Here are some examples of typical usage of the `uuid` module :

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

21.21 socketserver — A framework for network servers

Source code : [Lib/socketserver.py](#)

The `socketserver` module simplifies the task of writing network servers.

There are four basic concrete server classes :

class `socketserver.TCPServer` (*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke `server_bind()` and `server_activate()`. The other parameters are passed to the `BaseServer` base class.

class `socketserver.UDPServer` (*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for `TCPServer`.

class `socketserver.UnixStreamServer` (*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

```
class socketserver.UnixDatagramServer (server_address, RequestHandlerClass,
                                     bind_and_activate=True)
```

These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for *TCPServer*.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the *ForkingMixIn* and *ThreadingMixIn* mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the *BaseRequestHandler* class and overriding its *handle()* method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. It is recommended to use the server in a *with* statement. Then call the *handle_request()* or *serve_forever()* method of the server object to process one or many requests. Finally, call *server_close()* to close the socket (unless you used a *with* statement).

When inheriting from *ThreadingMixIn* for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The *ThreadingMixIn* class defines an attribute *daemon_threads*, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is *False*, meaning that Python will not exit until all threads created by *ThreadingMixIn* have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

21.21.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types :



Note that *UnixDatagramServer* derives from *UDPServer*, not from *UnixStreamServer* — the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

```
class socketserver.ForkingMixIn
class socketserver.ThreadingMixIn
```

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, *ThreadingUDPServer* is created as follows :

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in *UDPServer*. Setting the various attributes also changes the behavior of the underlying server mechanism.

ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support *fork()*.

```
class socketserver.ForkingTCPServer
class socketserver.ForkingUDPServer
class socketserver.ThreadingTCPServer
class socketserver.ThreadingUDPServer
```

These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from *BaseRequestHandler* and redefine its *handle()* method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses *StreamRequestHandler* or *DatagramRequestHandler*.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service « deaf » while one request is being handled – which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class *handle()* method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor *fork()* (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use *selectors* to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See *asyncore* for another way to manage this.

21.21.2 Objets Serveur

```
class socketserver.BaseServer (server_address, RequestHandlerClass)
```

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective *server_address* and *RequestHandlerClass* attributes.

```
fileno ()
```

Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to *selectors*, to allow monitoring multiple servers in the same process.

```
handle_request ()
```

Process a single request. This function calls the following methods in order : *get_request()*, *verify_request()*, and *process_request()*. If the user-provided *handle()* method of the handler class raises an exception, the server's *handle_error()* method will be called. If no request is received within *timeout* seconds, *handle_timeout()* will be called and *handle_request()* will return.

```
serve_forever (poll_interval=0.5)
```

Handle requests until an explicit *shutdown()* request. Poll for shutdown every *poll_interval* seconds. Ignores the *timeout* attribute. It also calls *service_actions()*, which may be used by a subclass or mixin to provide actions specific to a given service. For example, the *ForkingMixIn* class uses *service_actions()* to clean up zombie child processes.

Modifié dans la version 3.3 : Added *service_actions* call to the *serve_forever* method.

service_actions()

This is called in the `serve_forever()` loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

Nouveau dans la version 3.3.

shutdown()

Tell the `serve_forever()` loop to stop and wait until it does.

server_close()

Clean up the server. May be overridden.

address_family

The family of protocols to which the server's socket belongs. Common examples are `socket.AF_INET` and `socket.AF_UNIX`.

RequestHandlerClass

The user-provided request handler class; an instance of this class is created for each request.

server_address

The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the `socket` module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

socket

The socket object on which the server will listen for incoming requests.

The server classes support the following class variables :

allow_reuse_address

Whether the server will allow the reuse of an address. This defaults to `False`, and can be set in subclasses to change the policy.

request_queue_size

The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to `request_queue_size` requests. Once the queue is full, further requests from clients will get a « Connection denied » error. The default value is usually 5, but this can be overridden by subclasses.

socket_type

The type of socket used by the server; `socket.SOCK_STREAM` and `socket.SOCK_DGRAM` are two common values.

timeout

Timeout duration, measured in seconds, or `None` if no timeout is desired. If `handle_request()` receives no incoming requests within the timeout period, the `handle_timeout()` method is called.

There are various server methods that can be overridden by subclasses of base server classes like `TCPServer`; these methods aren't useful to external users of the server object.

finish_request(request, client_address)

Actually processes the request by instantiating `RequestHandlerClass` and calling its `handle()` method.

get_request()

Must accept a request from the socket, and return a 2-tuple containing the *new* socket object to be used to communicate with the client, and the client's address.

handle_error(request, client_address)

This function is called if the `handle()` method of a `RequestHandlerClass` instance raises an exception. The default action is to print the traceback to standard error and continue handling further requests.

Modifié dans la version 3.6 : Now only called for exceptions derived from the `Exception` class.

handle_timeout()

This function is called when the `timeout` attribute has been set to a value other than `None` and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

process_request (*request*, *client_address*)

Calls *finish_request()* to create an instance of the *RequestHandlerClass*. If desired, this function can create a new process or thread to handle the request; the *ForkingMixIn* and *ThreadingMixIn* classes do this.

server_activate ()

Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes *listen()* on the server's socket. May be overridden.

server_bind ()

Called by the server's constructor to bind the socket to the desired address. May be overridden.

verify_request (*request*, *client_address*)

Must return a Boolean value; if the value is *True*, the request will be processed, and if it's *False*, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns *True*.

Modifié dans la version 3.6 : Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling *server_close()*.

21.21.3 Request Handler Objects

class `socketserver.BaseRequestHandler`

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new *handle()* method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup ()

Called before the *handle()* method to perform any initialization actions required. The default implementation does nothing.

handle ()

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as *self.request*; the client address as *self.client_address*; and the server instance as *self.server*, in case it needs access to per-server information.

The type of *self.request* is different for datagram or stream services. For stream services, *self.request* is a socket object; for datagram services, *self.request* is a pair of string and socket.

finish ()

Called after the *handle()* method to perform any clean-up actions required. The default implementation does nothing. If *setup()* raises an exception, this function will not be called.

class `socketserver.StreamRequestHandler`

class `socketserver.DatagramRequestHandler`

These *BaseRequestHandler* subclasses override the *setup()* and *finish()* methods, and provide *self.rfile* and *self.wfile* attributes. The *self.rfile* and *self.wfile* attributes can be read or written, respectively, to get the request data or return data to the client.

The *rfile* attributes of both classes support the *io.BufferedIOBase* readable interface, and *DatagramRequestHandler.wfile* supports the *io.BufferedIOBase* writable interface.

Modifié dans la version 3.6 : *StreamRequestHandler.wfile* also supports the *io.BufferedIOBase* writable interface.

21.21.4 Exemples

socketserver.TCPServer Example

This is the server side :

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface) :

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the single `recv()` call in the first handler will just return what has been sent from the client in one `sendall()` call.

This is the client side :

```
import socket
import sys
```

(suite sur la page suivante)

(suite de la page précédente)

```

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent: {}".format(data))
print("Received: {}".format(received))

```

The output of the example should look something like this :

Serveur :

```

$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'

```

Client :

```

$ python TCPClient.py hello world with TCP
Sent: hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent: python is nice
Received: PYTHON IS NICE

```

socketserver.UDPServer Example

This is the server side :

```

import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:".format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)

```

(suite sur la page suivante)

(suite de la page précédente)

```

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()

```

This is the client side :

```

import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:      {}".format(data))
print("Received: {}".format(received))

```

The output of the example should look exactly like for the TCP server example.

Asynchronous Mixins

To build asynchronous handlers, use the *ThreadingMixin* and *ForkingMixin* classes.

An example for the *ThreadingMixin* class :

```

import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixin, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":

```

(suite sur la page suivante)

(suite de la page précédente)

```
# Port 0 means to select an arbitrary unused port
HOST, PORT = "localhost", 0

server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
with server:
    ip, port = server.server_address

    # Start a thread with the server -- that thread will then start one
    # more thread for each request
    server_thread = threading.Thread(target=server.serve_forever)
    # Exit the server thread when the main thread terminates
    server_thread.daemon = True
    server_thread.start()
    print("Server loop running in thread:", server_thread.name)

    client(ip, port, "Hello World 1")
    client(ip, port, "Hello World 2")
    client(ip, port, "Hello World 3")

server.shutdown()
```

The output of the example should look something like this :

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The *ForkingMixIn* class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support *fork()*.

21.22 http.server — HTTP servers

Source code : [Lib/http/server.py](http://lib/http/server.py)

This module defines classes for implementing HTTP servers (Web servers).

Avertissement : *http.server* is not recommended for production. It only implements only basic security checks.

One class, *HTTPServer*, is a *socketserver.TCPServer* subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this :

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class *http.server.HTTPServer* (*server_address*, *RequestHandlerClass*)

This class builds on the *TCPServer* class by storing the server address as instance variables named *server_name* and *server_port*. The server is accessible by the handler, typically through the handler's *server* instance variable.

The `HTTPServer` must be given a *RequestHandlerClass* on instantiation, of which this module provides three different variants :

class `http.server.BaseHTTPRequestHandler` (*request*, *client_address*, *server*)

This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). *BaseHTTPRequestHandler* provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method SPAM, the `do_SPAM()` method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the `__init__()` method.

BaseHTTPRequestHandler has the following instance variables :

client_address

Contains a tuple of the form (*host*, *port*) referring to the client's address.

server

Contains the server instance.

close_connection

Boolean that should be set before *handle_one_request()* returns, indicating if another request may be expected, or if the connection should be shut down.

requestline

Contains the string representation of the HTTP request line. The terminating CRLF is stripped. This attribute should be set by *handle_one_request()*. If no valid request line was processed, it should be set to the empty string.

command

Contains the command (request type). For example, 'GET'.

path

Contains the request path.

request_version

Contains the version string from the request. For example, 'HTTP/1.0'.

headers

Holds an instance of the class specified by the *MessageClass* class variable. This instance parses and manages the headers in the HTTP request. The *parse_headers()* function from *http.client* is used to parse the headers and it requires that the HTTP request provide a valid **RFC 2822** style header.

rfile

An *io.BufferedReader* input stream, ready to read from the start of the optional input data.

wfile

Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperability with HTTP clients.

Modifié dans la version 3.6 : This is an *io.BufferedReader* stream.

BaseHTTPRequestHandler has the following attributes :

server_version

Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form *name[/version]*. For example, 'BaseHTTP/0.2'.

sys_version

Contains the Python system version, in a form usable by the *version_string* method and the *server_version* class variable. For example, 'Python/1.4'.

error_message_format

Specifies a format string that should be used by *send_error()* method for building an error response to the client. The string is filled by default with variables from *responses* based on the status code that passed to *send_error()*.

error_content_type

Specifies the Content-Type HTTP header of error responses sent to the client. The default value is 'text/html'.

protocol_version

This specifies the HTTP protocol version used in responses. If set to 'HTTP/1.1', the server will permit HTTP persistent connections; however, your server *must* then include an accurate Content-Length header (using `send_header()`) in all of its responses to clients. For backwards compatibility, the setting defaults to 'HTTP/1.0'.

MessageClass

Specifies an `email.message.Message`-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to `http.client.HTTPMessage`.

responses

This attribute contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{code: (shortmessage, longmessage)}`. The *shortmessage* is usually used as the *message* key in an error response, and *longmessage* as the *explain* key. It is used by `send_response_only()` and `send_error()` methods.

A `BaseHTTPRequestHandler` instance has the following methods:

handle()

Calls `handle_one_request()` once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate `do_*()` methods.

handle_one_request()

This method will parse and dispatch the request to the appropriate `do_*()` method. You should never need to override it.

handle_expect_100()

When a HTTP/1.1 compliant server receives an `Expect: 100-continue` request header it responds back with a `100 Continue` followed by `200 OK` headers. This method can be overridden to raise an error if the server does not want the client to continue. For e.g. server can chose to send `417 Expectation Failed` as a response header and `return False`.

Nouveau dans la version 3.2.

send_error(code, message=None, explain=None)

Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as an optional, short, human readable description of the error. The *explain* argument can be used to provide more detailed information about the error; it will be formatted using the `error_message_format` attribute and emitted, after a complete set of headers, as the response body. The `responses` attribute holds the default values for *message* and *explain* that will be used if no value is provided; for unknown codes the default value for both is the string `???`. The body will be empty if the method is `HEAD` or the response code is one of the following: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

Modifié dans la version 3.4 : The error response includes a Content-Length header. Added the *explain* argument.

send_response(code, message=None)

Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by *Server* and *Date* headers. The values for these two headers are picked up from the `version_string()` and `date_time_string()` methods, respectively. If the server does not intend to send any other headers using the `send_header()` method, then `send_response()` should be followed by an `end_headers()` call.

Modifié dans la version 3.3 : Headers are stored to an internal buffer and `end_headers()` needs to be called explicitly.

send_header(keyword, value)

Adds the HTTP header to an internal buffer which will be written to the output stream when either `end_headers()` or `flush_headers()` is invoked. *keyword* should specify the header keyword, with *value* specifying its value. Note that, after the `send_header` calls are done, `end_headers()` MUST BE called in order to complete the operation.

Modifié dans la version 3.2 : Headers are stored in an internal buffer.

send_response_only (*code*, *message=None*)

Sends the response header only, used for the purposes when 100 Continue response is sent by the server to the client. The headers not buffered and sent directly the output stream. If the *message* is not specified, the HTTP message corresponding the response *code* is sent.

Nouveau dans la version 3.2.

end_headers ()

Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls *flush_headers* ().

Modifié dans la version 3.2 : The buffered headers are written to the output stream.

flush_headers ()

Finally send the headers to the output stream and flush the internal headers buffer.

Nouveau dans la version 3.3.

log_request (*code=''*, *size=''*)

Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

log_error (...)

Logs an error when a request cannot be fulfilled. By default, it passes the message to *log_message* (), so it takes the same arguments (*format* and additional values).

log_message (*format*, ...)

Logs an arbitrary message to *sys.stderr*. This is typically overridden to create custom error logging mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to *log_message* () are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string ()

Returns the server software's version string. This is a combination of the *server_version* and *sys_version* attributes.

date_time_string (*timestamp=None*)

Returns the date and time given by *timestamp* (which must be None or in the format returned by *time.time*()), formatted for a message header. If *timestamp* is omitted, it uses the current date and time.

The result looks like 'Sun, 06 Nov 1994 08:49:37 GMT'.

log_date_time_string ()

Returns the current date and time, formatted for logging.

address_string ()

Returns the client address.

Modifié dans la version 3.3 : Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class *http.server.SimpleHTTPRequestHandler* (*request*, *client_address*, *server*)

This class serves files from the current directory and below, directly mapping the directory structure to HTTP requests.

A lot of the work, such as parsing the request, is done by the base class *BaseHTTPRequestHandler*. This class implements the *do_GET* () and *do_HEAD* () functions.

The following are defined as class-level attributes of *SimpleHTTPRequestHandler* :

server_version

This will be "SimpleHTTP/" + *__version__*, where *__version__* is defined at the module level.

extensions_map

A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is considered to be application/octet-stream. The mapping is used case-insensitively, and so should contain only lower-cased keys.

The *SimpleHTTPRequestHandler* class defines the following methods :

do_HEAD()

This method serves the 'HEAD' request type : it sends the headers it would send for the equivalent GET request. See the `do_GET()` method for a more complete explanation of the possible headers.

do_GET()

The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named `index.html` or `index.htm` (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the `list_directory()` method. This method uses `os.listdir()` to scan the directory, and returns a 404 error response if the `listdir()` fails.

If the request was mapped to a file, it is opened and the contents are returned. Any `OSError` exception in opening the requested file is mapped to a 404, 'File not found' error. Otherwise, the content type is guessed by calling the `guess_type()` method, which in turn uses the `extensions_map` variable.

A 'Content-type:' header with the guessed content type is output, followed by a 'Content-Length:' header with the file's size and a 'Last-Modified:' header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with `text/` the file is opened in text mode; otherwise binary mode is used.

For example usage, see the implementation of the `test()` function invocation in the `http.server` module.

The `SimpleHTTPRequestHandler` class can be used in the following manner in order to create a very basic web-server serving files relative to the current directory :

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

`http.server` can also be invoked directly using the `-m` switch of the interpreter with a `port` number argument. Similar to the previous example, this serves files relative to the current directory :

```
python -m http.server 8000
```

By default, server binds itself to all interfaces. The option `-b/--bind` specifies a specific address to which it should bind. For example, the following command causes the server to bind to localhost only :

```
python -m http.server 8000 --bind 127.0.0.1
```

Nouveau dans la version 3.4 : `--bind` argument was introduced.

class `http.server.CGIHTTPRequestHandler` (*request, client_address, server*)

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in `SimpleHTTPRequestHandler`.

Note : CGI scripts run by the `CGIHTTPRequestHandler` class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used — the other common server configuration is to treat special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The `CGIHTTPRequestHandler` defines the following data member :

`cgi_directories`

This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The `CGIHTTPRequestHandler` defines the following method :

`do_POST()`

This method serves the 'POST' request type, only allowed for CGI scripts. Error 501, « Can only POST to CGI scripts », is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

`CGIHTTPRequestHandler` can be enabled in the command line by passing the `--cgi` option :

```
python -m http.server --cgi 8000
```

21.23 http.cookies — gestion d'état pour HTTP

Code source : Lib/http/cookies.py

Le module `http.cookies` définit des classes abstrayant le concept de témoin web (cookie), un mécanisme de gestion d'état pour HTTP. Il fournit une abstraction gérant des données textuelles et tout type de données sérialisable comme valeur de témoin.

Auparavant, le module appliquait strictement les règles d'analyse décrites dans les spécifications **RFC 2109** et **RFC 2068**. Entre temps, il a été découvert que Internet Explorer 3.0 ne suit pas les règles liées aux caractères précisées dans ces spécifications. De plus, plusieurs navigateurs et serveurs dans leur versions récentes ont assoupli les règles d'analyse quant à la gestion des témoins. En conséquence, les règles d'analyse utilisées sont un peu moins strictes que les spécifications initiales.

Les jeux de caractères `string.ascii_letters`, `string.digits` et `!#$%&'*+-.^_`|~` : définissent l'ensemble des caractères autorisés par ce module pour le nom du témoin (comme `key`).

Modifié dans la version 3.3 : Ajouté « : » comme caractère autorisé pour les noms de témoin.

Note : Quand un témoin invalide est rencontré, l'exception `CookieError` est levée. Si les données du témoin proviennent d'un navigateur il faut impérativement gérer les données invalides en attrapant `CookieError`.

exception `http.cookies.CookieError`

Exception levée pour cause d'incompatibilité avec la **RFC 2109**. Exemples : attributs incorrects, en-tête `Set-Cookie` incorrect, etc.

class `http.cookies.BaseCookie` (`[input]`)

Cette classe définit un dictionnaire dont les clés sont des chaînes de caractères et dont les valeurs sont des instances de `Morsel`. Notez qu'à l'assignation d'une valeur à une clé, la valeur est transformée en `Morsel` contenant la clé et la valeur.

Si l'argument `input` est donné, il est passé à la méthode `load()`.


```
class http.cookies.SimpleCookie([input])
```

This class derives from `BaseCookie` and overrides `value_decode()` and `value_encode()` to be the identity and `str()` respectively.

Voir aussi :

Module `http.cookiejar` Gestion de témoins HTTP pour *clients* web. Les modules `http.cookiejar` et `http.cookies` ne dépendent pas l'un de l'autre.

RFC 2109 - HTTP State Management Mechanism Spécification de gestion d'états implantée par ce module.

21.23.1 Objets *Cookie*

```
BaseCookie.value_decode(val)
```

Return a decoded value from a string representation. Return value can be any type. This method does nothing in `BaseCookie` — it exists so it can be overridden.

```
BaseCookie.value_encode(val)
```

Return an encoded value. *val* can be any type, but return value must be a string. This method does nothing in `BaseCookie` — it exists so it can be overridden.

Généralement, les méthodes `value_encode()` et `value_decode()` doivent être inverses l'une de l'autre, c'est-à-dire qu'en envoyant la sortie de l'un dans l'entrée de l'autre la valeur finale doit être égale à la valeur initiale.

```
BaseCookie.output(attrs=None, header='Set-Cookie:', sep='\r\n')
```

Renvoie une représentation textuelle compatible avec les en-têtes HTTP. *attrs* et **header* sont envoyés à la méthode `output()` de chaque classe `Morsel`. *sep* est le séparateur à utiliser pour joindre les valeurs d'en-têtes. Sa valeur par défaut est `'\r\n'` (CRLF).

```
BaseCookie.js_output(attrs=None)
```

Renvoie un extrait de code JavaScript qui, lorsque exécuté par un navigateur qui supporte le JavaScript, va fonctionner de la même manière que si les en-têtes HTTP avaient été envoyés.

attrs a la même signification que dans la méthode `output()`.

```
BaseCookie.load(rawdata)
```

Si *rawdata* est une chaîne de caractères, l'analyser comme étant un `HTTP_COOKIE` et ajouter les valeurs trouvées en tant que `Morsels`. S'il s'agit d'un dictionnaire, cela est équivalent à :

```
for k, v in rawdata.items():
    cookie[k] = v
```

21.23.2 Objets *Morsel*

```
class http.cookies.Morsel
```

Abstraction de paire clé / valeur, accompagnée d'attributs provenant de la spécification **RFC 2109**.

Les objets *Morsel* sont des objets compatibles dictionnaire, dont l'ensemble des clés est fixe et égal aux attributs **RFC 2109** valides, qui sont

- `expires`
- `path`
- `comment`
- `domain`
- `max-age`
- `secure`
- `version`
- `httponly`

L'attribut `httponly` spécifie que le témoin transféré dans les requêtes HTTP n'est pas accessible par le biais de JavaScript. Il s'agit d'une contremesure à certaines attaques de scripts inter-sites (XSS).

Les clés ne sont pas sensibles à la casse, leur valeur par défaut est `' '`.

Modifié dans la version 3.5 : Dorénavant, `__eq__()` prend en compte *key* et *value*.

Morsel.value

La valeur du témoin.

Obsolète depuis la version 3.5 : assigning to *value* ; use `set()` instead.

Morsel.coded_value

La valeur codée du témoin. C'est celle qui doit être transférée.

Obsolète depuis la version 3.5 : assigning to *coded_value* ; use `set()` instead.

Morsel.key

Le nom du témoin.

Obsolète depuis la version 3.5 : assigning to *key* ; use `set()` instead.

Morsel.set (*key*, *value*, *coded_value*)

Assigne les attributs *key*, *value* et *coded_value*.

Obsolète depuis la version 3.5 : The undocumented *LegalChars* parameter is ignored and will be removed in a future version.

Morsel.isReservedKey (*K*)

Renvoie si *K* est membre des clés d'un *Morsel*.

Morsel.output (*attrs=None*, *header='Set-Cookie: '*)

Renvoie une représentation textuelle du *Morsel* compatible avec les en-têtes HTTP. Par défaut, tous les attributs sont inclus, à moins que *attrs* ne soit renseigné. Dans ce cas la valeur doit être une liste d'attributs à utiliser. Par défaut, *header* a la valeur `"Set-Cookie: "`.

Morsel.js_output (*attrs=None*)

Renvoie un extrait de code JavaScript qui, lorsque exécuté par un navigateur qui supporte le JavaScript, va fonctionner de la même manière que si les en-têtes HTTP avaient été envoyés.

attrs a la même signification que dans la méthode `output()`.

Morsel.OutputString (*attrs=None*)

Renvoie une chaîne de caractères représentant le *Morsel*, nettoyé de son contexte HTTP ou JavaScript.

attrs a la même signification que dans la méthode `output()`.

Morsel.update (*values*)

Met à jour les valeurs du dictionnaire du *Morsel* avec les valeurs provenant du dictionnaire *values*. Lève une erreur si une des clés n'est pas un attribut **RFC 2109** valide.

Modifié dans la version 3.5 : une erreur est levée pour les clés invalides.

Morsel.copy (*value*)

Renvoie une copie superficielle de l'objet *Morsel*.

Modifié dans la version 3.5 : renvoie un objet *Morsel* au lieu d'un `dict`.

Morsel.setdefault (*key*, *value=None*)

Lève une erreur si la clé n'est pas un attribut **RFC 2109** valide, sinon fonctionne de la même manière que `dict.setdefault()`.

21.23.3 Exemple

L'exemple suivant montre comment utiliser le module `http.cookies`.

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\\\"Loves\\""; fudge=\\012;"')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\\\\"Loves\\""; fudge=\\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

21.24 `http.cookiejar` — Cookie handling for HTTP clients

Source code : <Lib/http/cookiejar.py>

The `http.cookiejar` module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data – *cookies* – to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by [RFC 2965](#) are handled. RFC 2965 handling is switched off by default. [RFC 2109](#) cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the “policy” in effect. Note that the great majority of cookies on the Internet are Netscape cookies. `http.cookiejar` attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the `max-age` and `port` cookie-attributes introduced with RFC 2965.

Note : The various named parameters found in *Set-Cookie* and *Set-Cookie2* headers (eg. `domain` and `expires`) are conventionally referred to as *attributes*. To distinguish them from Python attributes, the documentation for this module uses the term *cookie-attribute* instead.

The module defines the following exception :

exception `http.cookiejar.LoadError`

Instances of `FileCookieJar` raise this exception on failure to load cookies from a file. `LoadError` is a subclass of `OSError`.

Modifié dans la version 3.3 : `LoadError` was made a subclass of `OSError` instead of `IOError`.

The following classes are provided :

class `http.cookiejar.CookieJar` (*policy=None*)

policy is an object implementing the `CookiePolicy` interface.

The `CookieJar` class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. `CookieJar` instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

class `http.cookiejar.FileCookieJar` (*filename, delayload=None, policy=None*)

policy is an object implementing the `CookiePolicy` interface. For the other arguments, see the documentation for the corresponding attributes.

A `CookieJar` which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are **NOT** loaded from the named file until either the `load()` or `revert()` method is called. Subclasses of this class are documented in section *FileCookieJar subclasses and co-operation with web browsers*.

class `http.cookiejar.CookiePolicy`

This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

class `http.cookiejar.DefaultCookiePolicy` (*blocked_domains=None, allo-*
wed_domains=None, netscape=True,
rfc2965=False, rfc2109_as_netscape=None,
hide_cookie2=False, strict_domain=False,
strict_rfc2965_unverifiable=True,
strict_ns_unverifiable=False,
strict_ns_domain=DefaultCookiePolicy.DomainLiberal,
strict_ns_set_initial_dollar=False,
strict_ns_set_path=False)

Constructor arguments should be passed as keyword arguments only. *blocked_domains* is a sequence of domain

names that we never accept cookies from, nor return cookies to. *allowed_domains* if not *None*, this is a sequence of the only domains for which we accept and return cookies. For all other arguments, see the documentation for *CookiePolicy* and *DefaultCookiePolicy* objects.

DefaultCookiePolicy implements the standard accept / reject rules for Netscape and **RFC 2965** cookies. By default, **RFC 2109** cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or *rfc2109_as_netscape* is *True*, RFC 2109 cookies are “downgraded” by the *CookieJar* instance to Netscape cookies, by setting the *version* attribute of the *Cookie* instance to 0. *DefaultCookiePolicy* also provides some parameters to allow some fine-tuning of policy.

class `http.cookiejar.Cookie`

This class represents Netscape, **RFC 2109** and **RFC 2965** cookies. It is not expected that users of *http.cookiejar* construct their own *Cookie* instances. Instead, if necessary, call *make_cookies()* on a *CookieJar* instance.

Voir aussi :

Module *urllib.request* URL opening with automatic cookie handling.

Module *http.cookies* HTTP cookie classes, principally useful for server-side code. The *http.cookiejar* and *http.cookies* modules do not depend on each other.

https://curl.haxx.se/rfc/cookie_spec.html The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the “Netscape cookie protocol” implemented by all the major browsers (and *http.cookiejar*) only bears a passing resemblance to the one sketched out in *cookie_spec.html*.

RFC 2109 - HTTP State Management Mechanism Obsoleted by **RFC 2965**. Uses *Set-Cookie* with *version=1*.

RFC 2965 - HTTP State Management Mechanism The Netscape protocol with the bugs fixed. Uses *Set-Cookie2* in place of *Set-Cookie*. Not widely used.

<http://kristol.org/cookie/errata.html> Unfinished errata to **RFC 2965**.

RFC 2964 - Use of HTTP State Management

21.24.1 CookieJar and FileCookieJar Objects

CookieJar objects support the *iterator* protocol for iterating over contained *Cookie* objects.

CookieJar has the following methods :

CookieJar.add_cookie_header(request)

Add correct *Cookie* header to *request*.

If policy allows (ie. the *rfc2965* and *hide_cookie2* attributes of the *CookieJar*’s *CookiePolicy* instance are *true* and *false* respectively), the *Cookie2* header is also added when appropriate.

The *request* object (usually a *urllib.request.Request* instance) must support the methods *get_full_url()*, *get_host()*, *get_type()*, *unverifiable()*, *has_header()*, *get_header()*, *header_items()*, *add_unredirected_header()* and *origin_req_host* attribute as documented by *urllib.request*.

Modifié dans la version 3.3 : *request* object needs *origin_req_host* attribute. Dependency on a deprecated method *get_origin_req_host()* has been removed.

CookieJar.extract_cookies(response, request)

Extract cookies from HTTP *response* and store them in the *CookieJar*, where allowed by policy.

The *CookieJar* will look for allowable *Set-Cookie* and *Set-Cookie2* headers in the *response* argument, and store cookies as appropriate (subject to the *CookiePolicy.set_ok()* method’s approval).

The *response* object (usually the result of a call to *urllib.request.urlopen()*, or similar) should support an *info()* method, which returns an *email.message.Message* instance.

The `request` object (usually a `urllib.request.Request` instance) must support the methods `get_full_url()`, `get_host()`, `unverifiable()`, and `origin_req_host` attribute, as documented by `urllib.request`. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

Modifié dans la version 3.3 : `request` object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`CookieJar.set_policy(policy)`

Set the `CookiePolicy` instance to be used.

`CookieJar.make_cookies(response, request)`

Return sequence of `Cookie` objects extracted from `response` object.

See the documentation for `extract_cookies()` for the interfaces required of the `response` and `request` arguments.

`CookieJar.set_cookie_if_ok(cookie, request)`

Set a `Cookie` if policy says it's OK to do so.

`CookieJar.set_cookie(cookie)`

Set a `Cookie`, without checking with policy to see whether or not it should be set.

`CookieJar.clear([domain[, path[, name]])]`

Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that `domain` will be removed. If given two arguments, cookies belonging to the specified `domain` and URL `path` are removed. If given three arguments, then the cookie with the specified `domain`, `path` and `name` is removed.

Raises `KeyError` if no matching cookie exists.

`CookieJar.clear_session_cookies()`

Discard all session cookies.

Discards all contained cookies that have a true `discard` attribute (usually because they had either no `max-age` or `expires` cookie-attribute, or an explicit `discard` cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the `save()` method won't save session cookies anyway, unless you ask otherwise by passing a true `ignore_discard` argument.

`FileCookieJar` implements the following additional methods :

`FileCookieJar.save(filename=None, ignore_discard=False, ignore_expires=False)`

Save cookies to a file.

This base class raises `NotImplementedError`. Subclasses may leave this method unimplemented.

`filename` is the name of file in which to save cookies. If `filename` is not specified, `self.filename` is used (whose default is the value passed to the constructor, if any); if `self.filename` is `None`, `ValueError` is raised.

`ignore_discard` : save even cookies set to be discarded. `ignore_expires` : save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the `load()` or `revert()` methods.

`FileCookieJar.load(filename=None, ignore_discard=False, ignore_expires=False)`

Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for `save()`.

The named file must be in the format understood by the class, or `LoadError` will be raised. Also, `OSError` may be raised, for example if the file does not exist.

Modifié dans la version 3.3 : `IOError` était normalement levée, elle est maintenant un alias de `OSError`.

`FileCookieJar.revert(filename=None, ignore_discard=False, ignore_expires=False)`

Clear all cookies and reload cookies from a saved file.

`revert()` can raise the same exceptions as `load()`. If there is a failure, the object's state will not be altered.

FileCookieJar instances have the following public attributes :

`FileCookieJar.filename`

Filename of default file in which to keep cookies. This attribute may be assigned to.

`FileCookieJar.delayload`

If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A *CookieJar* object may ignore it. None of the *FileCookieJar* classes included in the standard library lazily loads cookies.

21.24.2 FileCookieJar subclasses and co-operation with web browsers

The following *CookieJar* subclasses are provided for reading and writing.

class `http.cookiejar.MozillaCookieJar` (*filename*, *delayload=None*, *policy=None*)

A *FileCookieJar* that can load from and save cookies to disk in the Mozilla `cookies.txt` file format (which is also used by the Lynx and Netscape browsers).

Note : This loses information about **RFC 2965** cookies, and also about newer or non-standard cookie-attributes such as `port`.

Avertissement : Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class `http.cookiejar.LWPCookieJar` (*filename*, *delayload=None*, *policy=None*)

A *FileCookieJar* that can load from and save cookies to disk in format compatible with the libwww-perl library's Set-Cookie3 file format. This is convenient if you want to store cookies in a human-readable file.

21.24.3 CookiePolicy Objects

Objects implementing the *CookiePolicy* interface have the following methods :

`CookiePolicy.set_ok` (*cookie*, *request*)

Return boolean value indicating whether cookie should be accepted from server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.extract_cookies()*.

`CookiePolicy.return_ok` (*cookie*, *request*)

Return boolean value indicating whether cookie should be returned to server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.add_cookie_header()*.

`CookiePolicy.domain_return_ok` (*domain*, *request*)

Return false if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from *domain_return_ok()* and *path_return_ok()* leaves all the work to *return_ok()*.

If *domain_return_ok()* returns true for the cookie domain, *path_return_ok()* is called for the cookie path. Otherwise, *path_return_ok()* and *return_ok()* are never called for that cookie domain. If *path_return_ok()* returns true, *return_ok()* is called with the *Cookie* object itself for a full check. Otherwise, *return_ok()* is never called for that cookie path.

Note that `domain_return_ok()` is called for every *cookie* domain, not just for the *request* domain. For example, the function might be called with both `".example.com"` and `"www.example.com"` if the request domain is `"www.example.com"`. The same goes for `path_return_ok()`.

The *request* argument is as documented for `return_ok()`.

`CookiePolicy.path_return_ok(path, request)`

Return false if cookies should not be returned, given cookie path.

See the documentation for `domain_return_ok()`.

In addition to implementing the methods above, implementations of the `CookiePolicy` interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

`CookiePolicy.netscape`

Implement Netscape protocol.

`CookiePolicy.rfc2965`

Implement **RFC 2965** protocol.

`CookiePolicy.hide_cookie2`

Don't add *Cookie2* header to requests (the presence of this header indicates to the server that we understand **RFC 2965** cookies).

The most useful way to define a `CookiePolicy` class is by subclassing from `DefaultCookiePolicy` and overriding some or all of the methods above. `CookiePolicy` itself may be used as a “null policy” to allow setting and receiving any and all cookies (this is unlikely to be useful).

21.24.4 DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both **RFC 2965** and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks :

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the `CookiePolicy` interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the `blocked_domains` constructor argument, and `blocked_domains()` and `set_blocked_domains()` methods (and the corresponding argument and methods for `allowed_domains`). If you set a whitelist, you can turn it off again by setting it to `None`.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, `"example.com"` matches a blacklist entry of `"example.com"`, but `"www.example.com"` does not. Domains that do start with a dot are matched by more specific domains too. For example, both `"www.example.com"` and `"www.coyote.example.com"` match `".example.com"` (but `"example.com"` itself does not). IP addresses are an exception, and must match exactly. For example, if `blocked_domains` contains `"192.168.1.2"` and `".168.1.2"`, `192.168.1.2` is blocked, but `193.168.1.2` is not.

`DefaultCookiePolicy` implements the following additional methods :

`DefaultCookiePolicy.blocked_domains()`

Return the sequence of blocked domains (as a tuple).

`DefaultCookiePolicy.set_blocked_domains(blocked_domains)`

Set the sequence of blocked domains.

`DefaultCookiePolicy.is_blocked(domain)`

Return whether *domain* is on the blacklist for setting or receiving cookies.

`DefaultCookiePolicy.allowed_domains()`

Return *None*, or the sequence of allowed domains (as a tuple).

`DefaultCookiePolicy.set_allowed_domains(allowed_domains)`

Set the sequence of allowed domains, or *None*.

`DefaultCookiePolicy.is_not_allowed(domain)`

Return whether *domain* is not on the whitelist for setting or receiving cookies.

`DefaultCookiePolicy` instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

`DefaultCookiePolicy.rfc2109_as_netscape`

If true, request that the `CookieJar` instance downgrade **RFC 2109** cookies (ie. cookies received in a `Set-Cookie` header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the `Cookie` instance to 0. The default value is *None*, in which case RFC 2109 cookies are downgraded if and only if **RFC 2965** handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

General strictness switches :

`DefaultCookiePolicy.strict_domain`

Don't allow sites to set two-component domains with country-code top-level domains like `.co.uk`, `.gov.uk`, `.co.nz`.etc. This is far from perfect and isn't guaranteed to work !

RFC 2965 protocol strictness switches :

`DefaultCookiePolicy.strict_rfc2965_unverifiable`

Follow **RFC 2965** rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability

Netscape protocol strictness switches :

`DefaultCookiePolicy.strict_ns_unverifiable`

Apply **RFC 2965** rules on unverifiable transactions even to Netscape cookies.

`DefaultCookiePolicy.strict_ns_domain`

Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

`DefaultCookiePolicy.strict_ns_set_initial_dollar`

Ignore cookies in `Set-Cookie :` headers that have names starting with '\$'.

`DefaultCookiePolicy.strict_ns_set_path`

Don't allow setting cookies whose path doesn't path-match request URI.

`strict_ns_domain` is a collection of flags. Its value is constructed by or-ing together (for example, `DomainStrictNoDots|DomainStrictNonDomain` means both flags are set).

`DefaultCookiePolicy.DomainStrictNoDots`

When setting cookies, the "host prefix" must not contain a dot (eg. `www.foo.bar.com` can't set a cookie for `.bar.com`, because `www.foo` contains a dot).

`DefaultCookiePolicy.DomainStrictNonDomain`

Cookies that did not explicitly specify a domain cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. `spam.example.com` won't be returned cookies from `example.com` that had no domain cookie-attribute).

`DefaultCookiePolicy.DomainRFC2965Match`

When setting cookies, require a full [RFC 2965](#) domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags :

`DefaultCookiePolicy.DomainLiberal`

Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

`DefaultCookiePolicy.DomainStrict`

Equivalent to `DomainStrictNoDots | DomainStrictNonDomain`.

21.24.5 Objets *Cookie*

Cookie instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the `max-age` and `expires` cookie-attributes contain equivalent information, and because [RFC 2109](#) cookies may be “downgraded” by [http.cookiejar](#) from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a *CookiePolicy* method. The class does not enforce internal consistency, so you should know what you're doing if you do that.

`Cookie.version`

Integer or *None*. Netscape cookies have *version* 0. [RFC 2965](#) and [RFC 2109](#) cookies have a *version* cookie-attribute of 1. However, note that [http.cookiejar](#) may “downgrade” RFC 2109 cookies to Netscape cookies, in which case *version* is 0.

`Cookie.name`

Cookie name (a string).

`Cookie.value`

Cookie value (a string), or *None*.

`Cookie.port`

String representing a port or a set of ports (eg. “80”, or “80,8080”), or *None*.

`Cookie.path`

Cookie path (a string, eg. `'/acme/rocket_launchers'`).

`Cookie.secure`

True if cookie should only be returned over a secure connection.

`Cookie.expires`

Integer expiry date in seconds since epoch, or *None*. See also the *is_expired()* method.

`Cookie.discard`

True if this is a session cookie.

`Cookie.comment`

String comment from the server explaining the function of this cookie, or *None*.

`Cookie.comment_url`

URL linking to a comment from the server explaining the function of this cookie, or *None*.

`Cookie.rfc2109`

True if this cookie was received as an [RFC 2109](#) cookie (ie. the cookie arrived in a *Set-Cookie* header, and the

value of the Version cookie-attribute in that header was 1). This attribute is provided because `http.cookiejar` may “downgrade” RFC 2109 cookies to Netscape cookies, in which case `version` is 0.

`Cookie.port_specified`

True if a port or set of ports was explicitly specified by the server (in the `Set-Cookie` / `Set-Cookie2` header).

`Cookie.domain_specified`

True if a domain was explicitly specified by the server.

`Cookie.domain_initial_dot`

True if the domain explicitly specified by the server began with a dot ('.').

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods :

`Cookie.has_nonstandard_attr(name)`

Return true if cookie has the named cookie-attribute.

`Cookie.get_nonstandard_attr(name, default=None)`

If cookie has the named cookie-attribute, return its value. Otherwise, return *default*.

`Cookie.set_nonstandard_attr(name, value)`

Set the value of the named cookie-attribute.

The `Cookie` class also defines the following method :

`Cookie.is_expired(now=None)`

True if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

21.24.6 Examples

The first example shows the most common usage of `http.cookiejar` :

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file) :

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of `DefaultCookiePolicy`. Turn on **RFC 2965** cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned :

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
```

(suite sur la page suivante)

(suite de la page précédente)

```
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

21.25 xmlrpc — Modules Serveur et Client XMLRPC

XML-RPC est une méthode pour appeler des procédures distantes utilisant XML via HTTP. XML-RPC permet à un client d'appeler des fonctions avec leurs arguments sur un serveur distant (désigné par une URI), et recevoir en retour des données structurées.

`xmlrpc` est un paquet rassemblant un client et un serveur XML-RPC. Ces modules sont :

- `xmlrpc.client`
- `xmlrpc.server`

21.26 xmlrpc.client — XML-RPC client access

Source code : [Lib/xmlrpc/client.py](#)

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

Avertissement : The `xmlrpc.client` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [Vulnérabilités XML](#).

Modifié dans la version 3.5 : For HTTPS URIs, `xmlrpc.client` now performs all the necessary certificate and host-name checks by default.

```
class xmlrpc.client.ServerProxy(uri, transport=None, encoding=None, verbose=False,
                                allow_none=False, use_datetime=False, use_builtin_types=False,
                                *, context=None)
```

Modifié dans la version 3.3 : The `use_builtin_types` flag was added.

A `ServerProxy` instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https : URLs and an internal `HTTP Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If `allow_none` is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a `TypeError`. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see <http://ontosys.com/xml-rpc/extensions.php> for a description. The `use_builtin_types` flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default. `datetime.datetime`, `bytes` and `bytearray` objects may be passed to calls. The obsolete `use_datetime` flag is similar to `use_builtin_types` but it applies only to date/time values.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication : `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP "Authorization" header, and sent to the remote server as part of the connection process when invoking an XML-RPC method.

You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, *context* may be `ssl.SSLContext` and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type) :

XML-RPC type	Type Python
<code>boolean</code>	<code>bool</code>
<code>int</code> , <code>i1</code> , <code>i2</code> , <code>i4</code> , <code>i8</code> or <code>biginteger</code>	<code>int</code> in range from -2147483648 to 2147483647. Values get the <code><int></code> tag.
<code>double</code> or <code>float</code>	<code>float</code> . Values get the <code><double></code> tag.
<code>string</code>	<code>str</code>
<code>array</code>	<code>list</code> or <code>tuple</code> containing conformable elements. Arrays are returned as <code>lists</code> .
<code>struct</code>	<code>dict</code> . Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their <code>__dict__</code> attribute is transmitted.
<code>dateTime.iso8601</code>	<code>DateTime</code> or <code>datetime.datetime</code> . Returned type depends on values of <code>use_builtin_types</code> and <code>use_datetime</code> flags.
<code>base64</code>	<code>Binary</code> , <code>bytes</code> or <code>bytearray</code> . Returned type depends on the value of the <code>use_builtin_types</code> flag.
<code>nil</code>	The <code>None</code> constant. Passing is allowed only if <code>allow_none</code> is true.
<code>bigdecimal</code>	<code>decimal.Decimal</code> . Returned type only.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special `Fault` instance, used to signal XML-RPC server errors, or `ProtocolError` used to signal an error in the HTTP/HTTPS transport layer. Both `Fault` and `ProtocolError` derive from a base class called `Error`. Note that the `xmlrpc` client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as `<`, `>`, and `&` will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use `bytes` or `bytearray` classes or the `Binary` wrapper class described below.

Server is retained as an alias for `ServerProxy` for backwards compatibility. New code should use `ServerProxy`.

Modifié dans la version 3.5 : Added the *context* argument.

Modifié dans la version 3.6 : Added support of type tags with prefixes (e.g. `ex:nil`). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics : `i1`, `i2`, `i8`, `biginteger`, `float` and `bigdecimal`. See <http://ws.apache.org/xmlrpc/types.html> for a description.

Voir aussi :

XML-RPC HOWTO A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification The official specification.

Unofficial XML-RPC Errata Fredrik Lundh's « unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at “best practices” to use when designing your own XML-RPC implementations. »

21.26.1 ServerProxy Objects

A *ServerProxy* instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a *Fault* or *ProtocolError* object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute :

`ServerProxy.system.listMethods()`

This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

`ServerProxy.system.methodSignature(name)`

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply « string, array ». If it expects three integers and returns a string, its signature is « string, int, int, int ».

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

`ServerProxy.system.methodHelp(name)`

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

Modifié dans la version 3.5 : Instances of *ServerProxy* support the *context manager* protocol for closing the underlying transport.

A working example follows. The server code :

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server :

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

21.26.2 Objets DateTime

class xmlrpc.client.DateTime

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a `datetime.datetime` instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code :

decode (*string*)

Accept a string as the instance's new time value.

encode (*out*)

Write the XML-RPC encoding of this *DateTime* item to the *out* stream object.

It also supports certain of Python's built-in operators through rich comparison and `__repr__()` methods.

A working example follows. The server code :

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server :

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

21.26.3 Binary Objects

class xmlrpc.client.Binary

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a *Binary* object is provided by an attribute :

data

The binary data encapsulated by the *Binary* instance. The data is provided as a *bytes* object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code :

decode (*bytes*)

Accept a base64 *bytes* object and decode it as the instance's new data.

encode (*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per [RFC 2045 section 6.8](#), which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through `__eq__()` and `__ne__()` methods.

Example usage of the binary objects. We're going to transfer an image over XMLRPC :

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

The client gets the image and saves it to a file :

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

21.26.4 Fault Objects

class `xmlrpc.client.Fault`

A *Fault* object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes :

faultCode

A string indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a *Fault* by returning a complex type object. The server code :

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server :

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
```

(suite sur la page suivante)

(suite de la page précédente)

```

    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)

```

21.26.5 ProtocolError Objects

class `xmlrpc.client.ProtocolError`

A *ProtocolError* object describes a protocol error in the underlying transport layer (such as a 404 “not found” error if the server named by the URI does not exist). It has the following attributes :

url

The URI or URL that triggered the error.

errcode

The error code.

errmsg

The error message or diagnostic string.

headers

A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we’re going to intentionally cause a *ProtocolError* by providing an invalid URI :

```

import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)

```

21.26.6 MultiCall Objects

The *MultiCall* object provides a way to encapsulate multiple calls to a remote server into a single request ¹.

class `xmlrpc.client.MultiCall` (*server*)

Create an object used to boxcar method calls. *server* is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the *MultiCall* object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a *generator*; iterating over this generator yields the individual results.

A usage example of this class follows. The server code :

```

from xmlrpc.server import SimpleXMLRPCServer

```

(suite sur la page suivante)

1. This approach has been first presented in a [discussion](#) on `xmlrpc.com`.

(suite de la page précédente)

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

The client code for the preceding server :

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

21.26.7 Convenience Functions

`xmlrpc.client.dumps` (*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

Convert *params* into an XML-RPC request, or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the `Fault` exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's `None` value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

`xmlrpc.client.loads` (*data*, *use_datetime=False*, *use_builtin_types=False*)

Convert an XML-RPC request or response into Python objects, a (*params*, *methodname*). *params* is a tuple of argument; *methodname* is a string, or None if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The *use_builtin_types* flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default.

The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Modifié dans la version 3.3 : The *use_builtin_types* flag was added.

21.26.8 Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how :

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport=transport)
print(server.examples.getStateName(41))
```

21.26.9 Example of Client and Server Usage

See *SimpleXMLRPCServer Example*.

Notes

21.27 xmlrpc.server — Basic XML-RPC servers

Source code : [Lib/xmlrpc/server.py](#)

The `xmlrpc.server` module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using *SimpleXMLRPCServer*, or embedded in a CGI environment, using *CGIXMLRPCRequestHandler*.

Avertissement : The `xmlrpc.server` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [Vulnérabilités XML](#).

```
class xmlrpc.server.SimpleXMLRPCServer(addr, requestHand-
                                     ler=SimpleXMLRPCRequestHandler, logRe-
                                     quests=True, allow_none=False, encoding=None,
                                     bind_and_activate=True, use_builtin_types=False)
```

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The `requestHandler` parameter should be a factory for request handler instances; it defaults to `SimpleXMLRPCRequestHandler`. The `addr` and `requestHandler` parameters are passed to the `socketserver.TCPServer` constructor. If `logRequests` is true (the default), requests will be logged; setting this parameter to false will turn off logging. The `allow_none` and `encoding` parameters are passed on to `xmlrpc.client` and control the XML-RPC responses that will be returned from the server. The `bind_and_activate` parameter controls whether `server_bind()` and `server_activate()` are called immediately by the constructor; it defaults to true. Setting it to false allows code to manipulate the `allow_reuse_address` class variable before the address is bound. The `use_builtin_types` parameter is passed to the `loads()` function and controls which types are processed when date/times values or binary data are received; it defaults to false.

Modifié dans la version 3.3 : The `use_builtin_types` flag was added.

```
class xmlrpc.server.CGIXMLRPCRequestHandler(allow_none=False, encoding=None,
                                           use_builtin_types=False)
```

Create a new instance to handle XML-RPC requests in a CGI environment. The `allow_none` and `encoding` parameters are passed on to `xmlrpc.client` and control the XML-RPC responses that will be returned from the server. The `use_builtin_types` parameter is passed to the `loads()` function and controls which types are processed when date/times values or binary data are received; it defaults to false.

Modifié dans la version 3.3 : The `use_builtin_types` flag was added.

```
class xmlrpc.server.SimpleXMLRPCRequestHandler
```

Create a new request handler instance. This request handler supports POST requests and modifies logging so that the `logRequests` parameter to the `SimpleXMLRPCServer` constructor parameter is honored.

21.27.1 SimpleXMLRPCServer Objects

The `SimpleXMLRPCServer` class is based on `socketserver.TCPServer` and provides a means of creating simple, stand alone XML-RPC servers.

```
SimpleXMLRPCServer.register_function(function, name=None)
```

Register a function that can respond to XML-RPC requests. If `name` is given, it will be the method name associated with `function`, otherwise `function.__name__` will be used. `name` can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

```
SimpleXMLRPCServer.register_instance(instance, allow_dotted_names=False)
```

Register an object which is used to expose method names which have not been registered using `register_function()`. If `instance` contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request. Its API is `def _dispatch(self, method, params)` (note that `params` does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as `func(*params)`, expanding the parameter list. The return value from `_dispatch()` is returned to the client as the result. If `instance` does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method.

If the optional `allow_dotted_names` argument is true and the instance does not have a `_dispatch()` method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

Avertissement : Enabling the `allow_dotted_names` option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

`SimpleXMLRPCServer.register_introspection_functions()`

Registers the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

`SimpleXMLRPCServer.register_multicall_functions()`

Registers the XML-RPC multicall function `system.multicall`.

`SimpleXMLRPCRequestHandler.rpc_paths`

An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 « no such page » HTTP error. If this tuple is empty, all paths will be considered valid. The default value is `('/', '/RPC2')`.

SimpleXMLRPCServer Example

Server code :

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(("localhost", 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

The following client code will call the methods made available by the preceding server :

```
import xmlrpc.client
```

(suite sur la page suivante)

(suite de la page précédente)

```
s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3)) # Returns 2**3 = 8
print(s.add(2,3)) # Returns 5
print(s.mul(5,2)) # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

The following example included in the `Lib/xmlrpc/server.py` module shows a server allowing dotted names and registering a multicall function.

Avertissement : Enabling the `allow_dotted_names` option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this example only within a secure, closed network.

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

This ExampleService demo can be invoked from the command line :

```
python -m xmlrpc.server
```

The client that interacts with the above server is included in `Lib/xmlrpc/client.py` :

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
```

(suite sur la page suivante)

(suite de la page précédente)

```
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

This client which interacts with the demo XMLRPC server can be invoked as :

```
python -m xmlrpc.client
```

21.27.2 CGIXMLRPCRequestHandler

The *CGIXMLRPCRequestHandler* class can be used to handle XML-RPC requests sent to Python CGI scripts.

CGIXMLRPCRequestHandler.**register_function** (*function*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with function, otherwise *function.__name__* will be used. *name* can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

CGIXMLRPCRequestHandler.**register_instance** (*instance*)

Register an object which is used to expose method names which have not been registered using *register_function()*. If instance contains a *_dispatch()* method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If instance does not have a *_dispatch()* method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

CGIXMLRPCRequestHandler.**register_introspection_functions** ()

Register the XML-RPC introspection functions *system.listMethods*, *system.methodHelp* and *system.methodSignature*.

CGIXMLRPCRequestHandler.**register_multicall_functions** ()

Register the XML-RPC multicall function *system.multicall*.

CGIXMLRPCRequestHandler.**handle_request** (*request_text=None*)

Handle an XML-RPC request. If *request_text* is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

Exemple :

```
class MyFuncs:
    def mul(self, x, y):
        return x * y

handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

21.27.3 Documenting XMLRPC server

These classes extend the above classes to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using *DocXMLRPCServer*, or embedded in a CGI environment, using *DocCGIXMLRPCRequestHandler*.

```
class xmlrpc.server.DocXMLRPCServer(addr, requestHandler=DocXMLRPCRequestHandler,
                                   logRequests=True, allow_none=False, encoding=None,
                                   bind_and_activate=True, use_builtin_types=True)
    Create a new server instance. All parameters have the same meaning as for SimpleXMLRPCServer; requestHandler defaults to DocXMLRPCRequestHandler.
    Modifié dans la version 3.3 : The use_builtin_types flag was added.
```

```
class xmlrpc.server.DocCGIXMLRPCRequestHandler
    Create a new instance to handle XML-RPC requests in a CGI environment.
```

```
class xmlrpc.server.DocXMLRPCRequestHandler
    Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the logRequests parameter to the DocXMLRPCServer constructor parameter is honored.
```

21.27.4 DocXMLRPCServer Objects

The *DocXMLRPCServer* class is derived from *SimpleXMLRPCServer* and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

```
DocXMLRPCServer.set_server_title(server_title)
    Set the title used in the generated HTML documentation. This title will be used inside the HTML « title » element.

DocXMLRPCServer.set_server_name(server_name)
    Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a « h1 » element.

DocXMLRPCServer.set_server_documentation(server_documentation)
    Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.
```

21.27.5 DocCGIXMLRPCRequestHandler

The *DocCGIXMLRPCRequestHandler* class is derived from *CGIXMLRPCRequestHandler* and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

```
DocCGIXMLRPCRequestHandler.set_server_title(server_title)
    Set the title used in the generated HTML documentation. This title will be used inside the HTML « title » element.

DocCGIXMLRPCRequestHandler.set_server_name(server_name)
    Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a « h1 » element.

DocCGIXMLRPCRequestHandler.set_server_documentation(server_documentation)
    Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.
```


21.28 `ipaddress` — IPv4/IPv6 manipulation library

Source code : [Lib/ipaddress.py](#)

`ipaddress` provides the capabilities to create, manipulate and operate on IPv4 and IPv6 addresses and networks.

The functions and classes in this module make it straightforward to handle various tasks related to IP addresses, including checking whether or not two hosts are on the same subnet, iterating over all hosts in a particular subnet, checking whether or not a string represents a valid IP address or network definition, and so on.

This is the full module API reference—for an overview and introduction, see `ipaddress-howto`.

Nouveau dans la version 3.3.

21.28.1 Convenience factory functions

The `ipaddress` module provides factory functions to conveniently create IP addresses, networks and interfaces :

`ipaddress.ip_address(address)`

Return an `IPv4Address` or `IPv6Address` object depending on the IP address passed as argument. Either IPv4 or IPv6 addresses may be supplied; integers less than 2^{32} will be considered to be IPv4 by default. A `ValueError` is raised if `address` does not represent a valid IPv4 or IPv6 address.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

`ipaddress.ip_network(address, strict=True)`

Return an `IPv4Network` or `IPv6Network` object depending on the IP address passed as argument. `address` is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied; integers less than 2^{32} will be considered to be IPv4 by default. `strict` is passed to `IPv4Network` or `IPv6Network` constructor. A `ValueError` is raised if `address` does not represent a valid IPv4 or IPv6 address, or if the network has host bits set.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

`ipaddress.ip_interface(address)`

Return an `IPv4Interface` or `IPv6Interface` object depending on the IP address passed as argument. `address` is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than 2^{32} will be considered to be IPv4 by default. A `ValueError` is raised if `address` does not represent a valid IPv4 or IPv6 address.

One downside of these convenience functions is that the need to handle both IPv4 and IPv6 formats means that error messages provide minimal information on the precise error, as the functions don't know whether the IPv4 or IPv6 format was intended. More detailed error reporting can be obtained by calling the appropriate version specific class constructors directly.

21.28.2 IP Addresses

Address objects

The `IPv4Address` and `IPv6Address` objects share a lot of common attributes. Some attributes that are only meaningful for IPv6 addresses are also implemented by `IPv4Address` objects, in order to make it easier to write code that handles both IP versions correctly. Address objects are *hashable*, so they can be used as keys in dictionaries.

```
class ipaddress.IPv4Address(address)
```

Construct an IPv4 address. An *AddressValueError* is raised if *address* is not a valid IPv4 address.

The following constitutes a valid IPv4 address :

1. A string in decimal-dot notation, consisting of four decimal integers in the inclusive range 0–255, separated by dots (e.g. 192 . 168 . 0 . 1). Each integer represents an octet (byte) in the address. Leading zeroes are tolerated only for values less than 8 (as there is no ambiguity between the decimal and octal interpretations of such strings).
2. An integer that fits into 32 bits.
3. An integer packed into a *bytes* object of length 4 (most significant octet first).

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

version

The appropriate version number : 4 for IPv4, 6 for IPv6.

max_prefixlen

The total number of bits in the address representation for this version : 32 for IPv4, 128 for IPv6.

The prefix defines the number of leading bits in an address that are compared to determine whether or not an address is part of a network.

compressed

exploded

The string representation in dotted decimal notation. Leading zeroes are never included in the representation. As IPv4 does not define a shorthand notation for addresses with octets set to zero, these two attributes are always the same as `str(addr)` for IPv4 addresses. Exposing these attributes makes it easier to write display code that can handle both IPv4 and IPv6 addresses.

packed

The binary representation of this address - a `bytes` object of the appropriate length (most significant octet first). This is 4 bytes for IPv4 and 16 bytes for IPv6.

```
reverse pointer
```

The name of the reverse DNS PTR record for the IP address, e.g. :

[illegible]

This is the name that could be used for performing a PTR lookup, not the resolved hostname itself.

Nouveau dans la version 3.5.

is multicast

True if the address is reserved for multicast use. See [RFC 3171](#) (for IPv4) or [RFC 2373](#) (for IPv6).

```
is_private
```

True if the address is allocated for private networks. See [iana-ipv4-special-registry](#) (for IPv4) or [iana-ipv6-special-registry](#) (for IPv6).

is_global

True if the address is allocated for public networks. See [iana-ipv4-special-registry](#) (for IPv4) or [iana-ipv6-special-registry](#) (for IPv6).

Nouveau dans la version 3.4.

is_unspecified

True if the address is unspecified. See [RFC 5735](#) (for IPv4) or [RFC 2373](#) (for IPv6).

is_reserved

True if the address is otherwise IETF reserved.

is_loopback

True if this is a loopback address. See [RFC 3330](#) (for IPv4) or [RFC 2373](#) (for IPv6).

is_link_local

True if the address is reserved for link-local usage. See [RFC 3927](#).

class `ipaddress.IPv6Address` (*address*)

Construct an IPv6 address. An `AddressValueError` is raised if *address* is not a valid IPv6 address.

The following constitutes a valid IPv6 address :

1. A string consisting of eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons. This describes an *exploded* (longhand) notation. The string can also be *compressed* (shorthand notation) by various means. See [RFC 4291](#) for details. For example, "0000:0000:0000:0000:0000:0abc:0007:0def" can be compressed to "::abc:7:def".
2. An integer that fits into 128 bits.
3. An integer packed into a `bytes` object of length 16, big-endian.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
```

compressed

The short form of the address representation, with leading zeroes in groups omitted and the longest sequence of groups consisting entirely of zeroes collapsed to a single empty group.

This is also the value returned by `str(addr)` for IPv6 addresses.

exploded

The long form of the address representation, with all leading zeroes and groups consisting entirely of zeroes included.

For the following attributes, see the corresponding documentation of the `IPv4Address` class :

packed**reverse_pointer****version****max_prefixlen****is_multicast****is_private****is_global****is_unspecified****is_reserved****is_loopback****is_link_local**

Nouveau dans la version 3.4 : `is_global`

is_site_local

True if the address is reserved for site-local usage. Note that the site-local address space has been deprecated by [RFC 3879](#). Use `is_private` to test if this address is in the space of unique local addresses as defined by [RFC 4193](#).

ipv4_mapped

For addresses that appear to be IPv4 mapped addresses (starting with `::FFFF/96`), this property will report the embedded IPv4 address. For any other address, this property will be `None`.

sixtofour

For addresses that appear to be 6to4 addresses (starting with `2002::/16`) as defined by [RFC 3056](#), this property will report the embedded IPv4 address. For any other address, this property will be `None`.

teredo

For addresses that appear to be Teredo addresses (starting with `2001::/32`) as defined by [RFC 4380](#), this property will report the embedded `(server, client)` IP address pair. For any other address, this property will be `None`.

Conversion to Strings and Integers

To interoperate with networking interfaces such as the `socket` module, addresses must be converted to strings or integers. This is handled using the `str()` and `int()` builtin functions :

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

Opérateurs

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Comparison operators

Address objects can be compared with the usual set of comparison operators. Some examples :

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
```

Arithmetic operators

Integers can be added to or subtracted from address objects. Some examples :

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
```

(suite sur la page suivante)

(suite de la page précédente)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```

21.28.3 IP Network definitions

The *IPv4Network* and *IPv6Network* objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a *mask* and a *network address*, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask 255.255.0 and the network address 192.168.1.0 consists of IP addresses in the inclusive range 192.168.1.0 to 192.168.1.255.

Prefix, net mask and host mask

There are several equivalent ways to specify IP network masks. A *prefix* /<nbits> is a notation that denotes how many high-order bits are set in the network mask. A *net mask* is an IP address with some number of high-order bits set. Thus the prefix /24 is equivalent to the net mask 255.255.255.0 in IPv4, or `ffff:ff00::` in IPv6. In addition, a *host mask* is the logical inverse of a *net mask*, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to /24 in IPv4 is 0.0.0.255.

Network objects

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between *IPv4Network* and *IPv6Network*, so to avoid duplication they are only documented for *IPv4Network*. Network objects are *hashable*, so they can be used as keys in dictionaries.

class `ipaddress.IPv4Network` (*address*, *strict=True*)

Construct an IPv4 network definition. *address* can be one of the following :

1. A string consisting of an IP address and an optional mask, separated by a slash (/). The IP address is the network address, and the mask can be either a single number, which means it's a *prefix*, or a string representation of an IPv4 address. If it's the latter, the mask is interpreted as a *net mask* if it starts with a non-zero field, or as a *host mask* if it starts with a zero field, with the single exception of an all-zero mask which is treated as a *net mask*. If no mask is provided, it's considered to be /32.
For example, the following *address* specifications are equivalent : 192.168.1.0/24, 192.168.1.0/255.255.255.0 and 192.168.1.0/0.0.0.255.
2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being /32.
3. An integer packed into a *bytes* object of length 4, big-endian. The interpretation is similar to an integer *address*.
4. A two-tuple of an address description and a netmask, where the address description is either a string, a 32-bits integer, a 4-bytes packed integer, or an existing IPv4Address object; and the netmask is either an integer representing the prefix length (e.g. 24) or a string representing the prefix mask (e.g. 255.255.255.0).

An *AddressValueError* is raised if *address* is not a valid IPv4 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv4 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise *TypeError* if the argument's IP version is incompatible to `self`.

Modifié dans la version 3.5 : Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

Refer to the corresponding attribute documentation in *IPv4Address*.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.

network_address

The network address for the network. The network address and the prefix length together uniquely define a network.

broadcast_address

The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

hostmask

The host mask, as an *IPv4Address* object.

netmask

The net mask, as an *IPv4Address* object.

with_prefixlen

compressed

exploded

A string representation of the network, with the mask in prefix notation.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

with_netmask

A string representation of the network, with the mask in net mask notation.

with_hostmask

A string representation of the network, with the mask in host mask notation.

num_addresses

The total number of addresses in the network.

prefixlen

Length of the network prefix, in bits.

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result.

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
```

overlaps(*other*)

True if this network is partly or wholly contained in *other* or *other* is wholly contained in this network.

address_exclude(*network*)

Computes the network definitions resulting from removing the given *network* from this one. Returns an iterator of network objects. Raises *ValueError* if *network* is not completely contained in this network.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets (*prefixlen_diff=1, new_prefix=None*)

The subnets that join to make the current network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be increased by. *new_prefix* is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns an iterator of network objects.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet (*prefixlen_diff=1, new_prefix=None*)

The supernet containing this network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be decreased by. *new_prefix* is the desired new prefix of the supernet; it must be smaller than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns a single network object.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

compare_networks (*other*)

Compare this network to *other*. In this comparison only the network addresses are considered; host bits aren't. Returns either -1, 0 or 1.

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

class `ipaddress.IPv6Network` (*address, strict=True*)

Construct an IPv6 network definition. *address* can be one of the following :

1. A string consisting of an IP address and an optional prefix length, separated by a slash (/). The IP address is the network address, and the prefix length must be a single number, the *prefix*. If no prefix length is provided, it's considered to be /128.

Note that currently expanded netmasks are not supported. That means `2001:db00::0/24` is a valid argument while `2001:db00::0/ffff:ff00::` not.

2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/128`.
3. An integer packed into a *bytes* object of length 16, big-endian. The interpretation is similar to an integer *address*.
4. A two-tuple of an address description and a netmask, where the address description is either a string, a 128-bits integer, a 16-bytes packed integer, or an existing `IPv6Address` object; and the netmask is an integer representing the prefix length.

An *AddressValueError* is raised if *address* is not a valid IPv6 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv6 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Modifié dans la version 3.5 : Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result.

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1, new_prefix=None*)

supernet(*prefixlen_diff=1, new_prefix=None*)

compare_networks(*other*)

Refer to the corresponding attribute documentation in *IPv4Network*.

is_site_local

These attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

Opérateurs

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

Itération

Network objects can be iterated to list all the addresses belonging to the network. For iteration, *all* hosts are returned, including unusable hosts (for usable hosts, use the `hosts()` method). An example :

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

Networks as containers of addresses

Network objects can act as containers of addresses. Some examples :

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

21.28.4 Interface objects

Interface objects are *hashable*, so they can be used as keys in dictionaries.

class `ipaddress.IPv4Interface` (*address*)

Construct an IPv4 interface. The meaning of *address* is as in the constructor of *IPv4Network*, except that arbitrary host addresses are always accepted.

IPv4Interface is a subclass of *IPv4Address*, so it inherits all the attributes from that class. In addition, the following attributes are available :

ip

The address (*IPv4Address*) without network information.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

The network (*IPv4Network*) this interface belongs to.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

A string representation of the interface with the mask in prefix notation.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

A string representation of the interface with the network as a net mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

A string representation of the interface with the network as a host mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class `ipaddress.IPv6Interface` (*address*)

Construct an IPv6 interface. The meaning of *address* is as in the constructor of *IPv6Network*, except that arbitrary host addresses are always accepted.

IPv6Interface is a subclass of *IPv6Address*, so it inherits all the attributes from that class. In addition, the following attributes are available :

ip

network

with_prefixlen

with_netmask

with_hostmask

Refer to the corresponding attribute documentation in *IPv4Interface*.

Opérateurs

Interface objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Interface objects can be compared with the usual set of logical operators.

For equality comparison (`==` and `!=`), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (`<`, `>`, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.

21.28.5 Other Module Level Functions

The module also provides the following module level functions :

`ipaddress.v4_int_to_packed(address)`

Represent an address as 4 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv4 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv4 IP address.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

`ipaddress.v6_int_to_packed(address)`

Represent an address as 16 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv6 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv6 IP address.

`ipaddress.summarize_address_range(first, last)`

Return an iterator of the summarized network range given the first and last IP addresses. *first* is the first *IPv4Address* or *IPv6Address* in the range and *last* is the last *IPv4Address* or *IPv6Address* in the range. A *TypeError* is raised if *first* or *last* are not IP addresses or are not of the same version. A *ValueError* is raised if *last* is not greater than *first* or if *first* address version is not 4 or 6.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...     ipaddress.IPv4Address('192.0.2.0'),
...     ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.
↪130/32')]
```

`ipaddress.collapse_addresses(addresses)`

Return an iterator of the collapsed *IPv4Network* or *IPv6Network* objects. *addresses* is an iterator of *IPv4Network* or *IPv6Network* objects. A *TypeError* is raised if *addresses* contains mixed version objects.

```
>>> [ipaddr for ipaddr in
...     ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
...     ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

`ipaddress.get_mixed_type_key(obj)`

Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default; they're fundamentally different, so the expression :

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

doesn't make sense. There are some times however, where you may wish to have *ipaddress* sort these anyway. If you need to do this, you can use this function as the *key* argument to *sorted()*.
obj is either a network or address object.

21.28.6 Custom Exceptions

To support more specific error reporting from class constructors, the module defines the following exceptions :

exception `ipaddress.AddressValueError` (*ValueError*)

Any value error related to the address.

exception `ipaddress.NetmaskValueError` (*ValueError*)

Any value error related to the net mask.

Les modules documentés dans ce chapitre implémentent divers algorithmes ou interfaces principalement utiles pour les applications multimédia. Ils peuvent ne pas être disponibles sur votre installation. En voici un aperçu :

22.1 `audioloop` — Manipulation de données audio brutes

The `audioloop` module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16, 24 or 32 bits wide, stored in *bytes-like objects*. All scalar items are integers, unless specified otherwise.

Modifié dans la version 3.4 : Support for 24-bit samples was added. All functions now accept any *bytes-like object*. String input now results in an immediate error.

This module provides support for a-LAW, u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions :

exception `audioloop.error`

This exception is raised on all errors, such as unknown number of bytes per sample, etc.

`audioloop.add(fragment1, fragment2, width)`

Return a fragment which is the addition of the two samples passed as parameters. *width* is the sample width in bytes, either 1, 2, 3 or 4. Both fragments should have the same length. Samples are truncated in case of overflow.

`audioloop.adpcm2lin(adpcmfragment, width, state)`

Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of `lin2adpcm()` for details on ADPCM coding. Return a tuple (*sample*, *newstate*) where the sample has the width specified in *width*.

`audioop.alaw2lin (fragment, width)`

Convert sound fragments in a-LAW encoding to linearly encoded sound fragments. a-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

`audioop.avg (fragment, width)`

Return the average over all samples in the fragment.

`audioop.avgpp (fragment, width)`

Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

`audioop.bias (fragment, width, bias)`

Return a fragment that is the original fragment with a bias added to each sample. Samples wrap around in case of overflow.

`audioop.byteswap (fragment, width)`

« Byteswap » all samples in a fragment and returns the modified fragment. Converts big-endian samples to little-endian and vice versa.

Nouveau dans la version 3.4.

`audioop.cross (fragment, width)`

Return the number of zero crossings in the fragment passed as an argument.

`audioop.findfactor (fragment, reference)`

Return a factor *F* such that `rms (add (fragment, mul (reference, -F)))` is minimal, i.e., return the factor with which you should multiply *reference* to make it match as well as possible to *fragment*. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to `len (fragment)`.

`audioop.findfit (fragment, reference)`

Try to match *reference* as well as possible to a portion of *fragment* (which should be the longer fragment). This is (conceptually) done by taking slices out of *fragment*, using `findfactor ()` to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple (*offset*, *factor*) where *offset* is the (integer) offset into *fragment* where the optimal match started and *factor* is the (floating-point) factor as per `findfactor ()`.

`audioop.findmax (fragment, length)`

Search *fragment* for a slice of length *length* samples (not bytes!) with maximum energy, i.e., return *i* for which `rms (fragment [i*2:(i+length)*2])` is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to `len (fragment)`.

`audioop.getsample (fragment, width, index)`

Return the value of sample *index* from the fragment.

`audioop.lin2adpcm (fragment, width, state)`

Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

state is a tuple containing the state of the coder. The coder returns a tuple (*adpcmfrag*, *newstate*), and the *newstate* should be passed to the next call of `lin2adpcm ()`. In the initial call, *None* can be passed as the state. *adpcmfrag* is the ADPCM coded fragment packed 2 4-bit values per byte.

`audioop.lin2alaw (fragment, width)`

Convert samples in the audio fragment to a-LAW encoding and return this as a bytes object. a-LAW is an audio encoding format whereby you get a dynamic range of about 13 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

`audioop.lin2lin (fragment, width, newwidth)`

Convert samples between 1-, 2-, 3- and 4-byte formats.

Note : In some audio formats, such as .WAV files, 16, 24 and 32 bit samples are signed, but 8 bit samples are unsigned. So when converting to 8 bit wide samples for these formats, you need to also add 128 to the result :

```
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

The same, in reverse, has to be applied when converting from 8 to 16, 24 or 32 bit width samples.

`audioop.lin2ulaw(fragment, width)`

Convert samples in the audio fragment to u-LAW encoding and return this as a bytes object. u-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

`audioop.max(fragment, width)`

Return the maximum of the *absolute value* of all samples in a fragment.

`audioop.maxpp(fragment, width)`

Return the maximum peak-peak value in the sound fragment.

`audioop.minmax(fragment, width)`

Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

`audioop.mul(fragment, width, factor)`

Return a fragment that has all samples in the original fragment multiplied by the floating-point value *factor*. Samples are truncated in case of overflow.

`audioop.ratecv(fragment, width, nchannels, inrate, outrate, state[, weightA[, weightB]])`

Convert the frame rate of the input fragment.

state is a tuple containing the state of the converter. The converter returns a tuple (*newfragment*, *newstate*), and *newstate* should be passed to the next call of `ratecv()`. The initial call should pass *None* as the state.

The *weightA* and *weightB* arguments are parameters for a simple digital filter and default to 1 and 0 respectively.

`audioop.reverse(fragment, width)`

Reverse the samples in a fragment and returns the modified fragment.

`audioop.rms(fragment, width)`

Return the root-mean-square of the fragment, i.e. $\sqrt{\sum(S_i^2)/n}$.

C'est une mesure de la puissance dans un signal audio.

`audioop.tomono(fragment, width, lfactor, rfactor)`

Convert a stereo fragment to a mono fragment. The left channel is multiplied by *lfactor* and the right channel by *rfactor* before adding the two channels to give a mono signal.

`audioop.tostereo(fragment, width, lfactor, rfactor)`

Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by *lfactor* and right channel samples by *rfactor*.

`audioop.ulaw2lin(fragment, width)`

Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

Note that operations such as `mul()` or `max()` make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that :

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
```

(suite sur la page suivante)

(suite de la page précédente)

```
rsample = audioop.tomono(sample, width, 0, 1)
lsample = audioop.mul(lsample, width, lfactor)
rsample = audioop.mul(rsample, width, rfactor)
lsample = audioop.tostereo(lsample, width, 1, 0)
rsample = audioop.tostereo(rsample, width, 0, 1)
return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the *initial* state (the one you passed to `lin2adpcm()`) along to the decoder, not the final state (as returned by the coder). If you want to use `struct.Struct` to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The `find*()` routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample :

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)      # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #                             out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

22.2 aifc — Lis et écrit dans les fichiers AIFF et AIFC

Code source : [Lib/aifc.py](#)

This module provides support for reading and writing AIFF and AIFF-C files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIFF-C is a newer version of the format that includes the ability to compress the audio data.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of `nchannels * samplesize` bytes, and a second's worth of audio consists of `nchannels * samplesize * framerate` bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes (176,400 bytes).

Le module `aifc` définit les fonctions suivantes :

`aifc.open(file, mode=None)`

Open an AIFF or AIFF-C file and return an object instance with methods that are described below. The argument *file* is either a string naming a file or a *file object*. *mode* must be 'r' or 'rb' when the file must be opened for reading, or 'w' or 'wb' when the file must be opened for writing. If omitted, *file.mode* is used if it exists, otherwise 'rb' is used. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use `writeframesraw()` and `setnframes()`. The `open()` function may be used in a `with` statement. When the `with` block completes, the `close()` method is called.

Modifié dans la version 3.4 : La prise en charge de l'instruction `with` a été ajoutée.

Objects returned by `open()` when a file is opened for reading have the following methods :

`aifc.getnchannels()`

Return the number of audio channels (1 for mono, 2 for stereo).

`aifc.getsampwidth()`

Donne la taille en octets des échantillons, individuellement.

`aifc.getframerate()`

Return the sampling rate (number of audio frames per second).

`aifc.getnframes()`

Donne le nombre de trames (*frames*) audio du fichier.

`aifc.getcomptype()`

Return a bytes array of length 4 describing the type of compression used in the audio file. For AIFF files, the returned value is `b'NONE'`.

`aifc.getcompname()`

Return a bytes array convertible to a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is `b'not compressed'`.

`aifc.getparams()`

Renvoie une `namedtuple()` (`nchannels`, `sampwidth`, `framerate`, `nframes`, `comptype`, `compname`), équivalent à la sortie des méthodes `get*()`.

`aifc.getmarkers()`

Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

`aifc.getmark(id)`

Return the tuple as described in `getmarkers()` for the mark with the given *id*.

`aifc.readframes(nframes)`

Read and return the next *nframes* frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

`aifc.rewind()`

Rewind the read pointer. The next `readframes()` will start from the beginning.

`aifc.setpos(pos)`

Va à la trame de numéro donné.

`aifc.tell()`

Donne le numéro de la trame courante.

`aifc.close()`

Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by `open()` when a file is opened for writing have all the above methods, except for `readframes()` and `setpos()`. In addition the following methods exist. The `get*()` methods can only be called after the corresponding

`set*()` methods have been called. Before the first `writeframes()` or `writeframesraw()`, all parameters except for the number of frames must be filled in.

`aifc.aiff()`

Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in `'.aiff'` in which case the default is an AIFF file.

`aifc.aifc()`

Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in `'.aiff'` in which case the default is an AIFF file.

`aifc.setnchannels(nchannels)`

Définit le nombre de canaux du fichier audio.

`aifc.setsampwidth(width)`

Définit la taille en octets des échantillons audio.

`aifc.setframerate(rate)`

Specify the sampling frequency in frames per second.

`aifc.setnframes(nframes)`

Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

`aifc.setcomptype(type, name)`

Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type as a bytes array, the type parameter should be a bytes array of length 4. Currently the following compression types are supported : `b'NONE'`, `b'ULAW'`, `b'ALAW'`, `b'G722'`.

`aifc.setparams(nchannels, sampwidth, framerate, comptype, compname)`

Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This means that it is possible to use the result of a `getparams()` call as argument to `setparams()`.

`aifc.setmark(id, pos, name)`

Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called at any time before `close()`.

`aifc.tell()`

Return the current write position in the output file. Useful in combination with `setmark()`.

`aifc.writeframes(data)`

Write data to the output file. This method can only be called after the audio file parameters have been set.

Modifié dans la version 3.4 : N'importe quel *bytes-like object* est maintenant accepté.

`aifc.writeframesraw(data)`

Like `writeframes()`, except that the header of the audio file is not updated.

Modifié dans la version 3.4 : N'importe quel *bytes-like object* est maintenant accepté.

`aifc.close()`

Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.

22.3 sunau — Read and write Sun AU files

Code source : [Lib/sunau.py](#)

The `sunau` module provides a convenient interface to the Sun AU sound format. Note that this module is interface-compatible with the modules `aifc` and `wave`.

An audio file consists of a header followed by the data. The fields of the header are :

Field	Sommaire
magic word	The four bytes <code>.snd</code> .
header size	Size of the header, including info, in bytes.
data size	Physical size of the data, in bytes.
encoding	Indicates how the audio samples are encoded.
sample rate	The sampling rate.
# of channels	The number of channels in the samples.
info	ASCII string giving a description of the audio file (padded with null bytes).

Apart from the info field, all header fields are 4 bytes in size. They are all 32-bit unsigned integers encoded in big-endian byte order.

The `sunau` module defines the following functions :

`sunau.open(file, mode)`

If *file* is a string, open the file by that name, otherwise treat it as a seekable file-like object. *mode* can be any of

'r' Mode lecture seule.

'w' Mode écriture seule.

Note that it does not allow read/write files.

A *mode* of 'r' returns an `AU_read` object, while a *mode* of 'w' or 'wb' returns an `AU_write` object.

`sunau.openfp(file, mode)`

Un synonyme de `open()`, maintenu pour la rétrocompatibilité.

The `sunau` module defines the following exception :

exception `sunau.Error`

An error raised when something is impossible because of Sun AU specs or implementation deficiency.

The `sunau` module defines the following data items :

`sunau.AUDIO_FILE_MAGIC`

An integer every valid Sun AU file begins with, stored in big-endian form. This is the string `.snd` interpreted as an integer.

`sunau.AUDIO_FILE_ENCODING_MULAW_8`

`sunau.AUDIO_FILE_ENCODING_LINEAR_8`

`sunau.AUDIO_FILE_ENCODING_LINEAR_16`

`sunau.AUDIO_FILE_ENCODING_LINEAR_24`

`sunau.AUDIO_FILE_ENCODING_LINEAR_32`

`sunau.AUDIO_FILE_ENCODING_ALAW_8`

Values of the encoding field from the AU header which are supported by this module.

`sunau.AUDIO_FILE_ENCODING_FLOAT`

`sunau.AUDIO_FILE_ENCODING_DOUBLE`

`sunau.AUDIO_FILE_ENCODING_ADPCM_G721`

```
sunau.AUDIO_FILE_ENCODING_ADPCM_G722
sunau.AUDIO_FILE_ENCODING_ADPCM_G723_3
sunau.AUDIO_FILE_ENCODING_ADPCM_G723_5
```

Additional known values of the encoding field from the AU header, but which are not supported by this module.

22.3.1 AU_read Objects

AU_read objects, as returned by *open()* above, have the following methods :

AU_read.close()

Close the stream, and make the instance unusable. (This is called automatically on deletion.)

AU_read.getnchannels()

Returns number of audio channels (1 for mono, 2 for stereo).

AU_read.getsampwidth()

Renvoie la largeur de l'échantillon en octets.

AU_read.getframerate()

Renvoie la fréquence d'échantillonnage.

AU_read.getnframes()

Renvoie le nombre de trames audio.

AU_read.getcomptype()

Returns compression type. Supported compression types are 'ULAW', 'ALAW' and 'NONE'.

AU_read.getcompname()

Human-readable version of *getcomptype()*. The supported types have the respective names 'CCITT G.711 u-law', 'CCITT G.711 A-law' and 'not compressed'.

AU_read.getparams()

Renvoie une *namedtuple()* (nchannels, sampwidth, framerate, nframes, comptype, compname), équivalent à la sortie des méthodes *get*()*.

AU_read.readframes(n)

Reads and returns at most *n* frames of audio, as a *bytes* object. The data will be returned in linear format. If the original data is in u-LAW format, it will be converted.

AU_read.rewind()

Remet le pointeur de fichier au début du flux audio.

Les deux fonctions suivantes utilisent le vocabulaire « position ». Ces positions sont compatible entre elles, la « position » de l'un est compatible avec la « position » de l'autre. Cette position est dépendante de l'implémentation.

AU_read.setpos(pos)

Set the file pointer to the specified position. Only values returned from *tell()* should be used for *pos*.

AU_read.tell()

Return current file pointer position. Note that the returned value has nothing to do with the actual position in the file.

The following two functions are defined for compatibility with the *aifc*, and don't do anything interesting.

AU_read.getmarkers()

Renvoie None.

AU_read.getmark(id)

Lève une erreur.

22.3.2 AU_write Objects

AU_write objects, as returned by `open()` above, have the following methods :

`AU_write.setnchannels(n)`
Définit le nombre de canaux.

`AU_write.setsampwidth(n)`
Set the sample width (in bytes.)
Modifié dans la version 3.4 : Added support for 24-bit samples.

`AU_write.setframerate(n)`
Set the frame rate.

`AU_write.setnframes(n)`
Set the number of frames. This can be later changed, when and if more frames are written.

`AU_write.setcomptype(type, name)`
Set the compression type and description. Only 'NONE' and 'ULAW' are supported on output.

`AU_write.setparams(tuple)`
The *tuple* should be (nchannels, sampwidth, framerate, nframes, comptype, compname), with values valid for the `set*()` methods. Set all parameters.

`AU_write.tell()`
Return current position in the file, with the same disclaimer for the `AU_read.tell()` and `AU_read.setpos()` methods.

`AU_write.writeframesraw(data)`
Écrit les trames audio sans corriger *nframes*.
Modifié dans la version 3.4 : N'importe quel *bytes-like object* est maintenant accepté.

`AU_write.writeframes(data)`
Write audio frames and make sure *nframes* is correct.
Modifié dans la version 3.4 : N'importe quel *bytes-like object* est maintenant accepté.

`AU_write.close()`
Make sure *nframes* is correct, and close the file.
This method is called upon deletion.

Note that it is invalid to set any parameters after calling `writeframes()` or `writeframesraw()`.

22.4 wave — Lecture et écriture des fichiers WAV

Code source : [Lib/wave.py](#)

Le module `wave` fournit une interface pratique pour le format de son WAV. Il ne gère pas la compression ni la décompression, mais gère le mono et le stéréo.

Le module `wave` définit la fonction et l'exception suivante :

`wave.open(file, mode=None)`
Si *file* est une chaîne de caractères, ouvre le fichier sous ce nom, sinon, il est traité comme un objet de type fichier.
mode peut être :
'rb' Mode lecture seule.
'wb' Mode écriture seule.

Notez que ce module ne permet pas de manipuler des fichiers WAV en lecture/écriture.

Un *mode* `'rb'` renvoie un objet `Wave_read`, alors qu'un *mode* `'wb'` renvoie un objet `Wave_write`. Si *mode* est omis et qu'un objet de type fichier est donné au paramètre *file*, `file.mode` est utilisé comme valeur par défaut pour *mode*.

Si vous donnez un objet de type fichier, l'objet *wave* ne le ferme pas lorsque sa méthode `close()` est appelée car c'est l'appelant qui est responsable de la fermeture.

The `open()` function may be used in a `with` statement. When the `with` block completes, the `Wave_read.close()` or `Wave_write.close()` method is called.

Modifié dans la version 3.4 : Ajout de la gestion des fichiers non navigables.

`wave.openfp(file, mode)`

Un synonyme de `open()`, maintenu pour la rétrocompatibilité.

exception `wave.Error`

Une erreur est levée lorsque quelque chose est impossible car elle enfreint la spécification WAV ou rencontre un problème d'implémentation.

22.4.1 Objets `Wave_read`

Les objets `Wave_read`, tels qu'ils sont renvoyés par `open()`, ont les méthodes suivantes :

`Wave_read.close()`

Ferme le flux s'il a été ouvert par *wave* et rend l'instance inutilisable. Ceci est appelé automatiquement lorsque l'objet est détruit.

`Wave_read.getnchannels()`

Renvoie le nombre de canaux audio (1 pour mono, 2 pour stéréo).

`Wave_read.getsampwidth()`

Renvoie la largeur de l'échantillon en octets.

`Wave_read.getframerate()`

Renvoie la fréquence d'échantillonnage.

`Wave_read.getnframes()`

Renvoie le nombre de trames audio.

`Wave_read.getcomptype()`

Renvoie le type de compression (`'NONE'` est le seul type géré).

`Wave_read.getcompname()`

Version compréhensible de `getcomptype()`. Généralement, `'not compressed'` équivaut à `'NONE'`.

`Wave_read.getparams()`

Renvoie une *namedtuple* (`nchannels`, `sampwidth`, `framerate`, `nframes`, `comptype`, `compname`), équivalent à la sortie des méthodes `get*`.

`Wave_read.readframes(n)`

Lit et renvoie au plus *n* trames audio, sous forme d'objet *bytes*.

`Wave_read.rewind()`

Remet le pointeur de fichier au début du flux audio.

Les deux méthodes suivantes sont définies pour la compatibilité avec le module *aifc*; elles ne font rien d'intéressant.

`Wave_read.getmarkers()`

Renvoie `None`.

`Wave_read.getmark(id)`

Lève une erreur.

Les deux fonctions suivantes utilisent le vocabulaire « position ». Ces positions sont compatible entre elles, la « position » de l'un est compatible avec la « position » de l'autre. Cette position est dépendante de l'implémentation.

`Wave_read.setpos(pos)`

Place le pointeur du fichier sur la position spécifiée.

`Wave_read.tell()`

Renvoie la position actuelle du pointeur du fichier.

22.4.2 Objets Wave_write

Pour les flux de sortie navigables, l'en-tête `wave` est automatiquement mis à jour pour refléter le nombre de trames réellement écrites. Pour les flux non indexables, la valeur `nframes` doit être précise lorsque la première trame est écrite. Une valeur précise de `nframes` peut être obtenue en appelant les méthodes `setnframes()` ou `setparams()` en passant en paramètre le nombre de trames qui seront écrites avant que `close()` soit appelé puis en utilisant la méthode `writeframesraw()` pour écrire les trames audio, ou en appelant la méthode `writeframes()` avec toutes les trames audio. Dans ce dernier cas, la méthode `writeframes()` calcule le nombre de trames dans le flux audio et définit `nframes` en conséquence avant d'écrire les données des trames.

Les objets `Wave_write`, tels qu'ils sont renvoyés par `open()`, ont les méthodes suivantes :

Modifié dans la version 3.4 : Ajout de la gestion des fichiers non navigables.

`Wave_write.close()`

Assurez-vous que `nframes` soit correct et fermez le fichier s'il a été ouvert par `wave`. Cette méthode est appelée à la destruction de l'objet. Il lève une erreur si le flux de sortie n'est pas navigable et si `nframes` ne correspond pas au nombre de trames réellement écrites.

`Wave_write.setnchannels(n)`

Définit le nombre de canaux.

`Wave_write.setsampwidth(n)`

Définit la largeur de l'échantillon à `n` octets.

`Wave_write.setframerate(n)`

Définit la fréquence des trames à `n`.

Modifié dans la version 3.2 : Un paramètre non-entier passé à cette méthode est arrondi à l'entier le plus proche.

`Wave_write.setnframes(n)`

Définit le nombre de trames à `n`. Cela sera modifié ultérieurement si le nombre de trames réellement écrites est différent (la tentative de mise à jour générera une erreur si le flux de sortie n'est pas indexable).

`Wave_write.setcomptype(type, name)`

Définit le type de compression et la description. Pour le moment, seul le type de compression `NONE` est géré, c'est-à-dire aucune compression.

`Wave_write.setparams(tuple)`

Le `tuple` doit être (`nchannels`, `sampwidth`, `framerate`, `nframes`, `comptype`, `compname`), avec des valeurs valides pour les méthodes `set*()`. Tous les paramètres sont obligatoires et doivent être définis.

`Wave_write.tell()`

Renvoie la position actuelle dans le fichier, avec les mêmes réserves que pour les méthodes `Wave_read.tell()` et `Wave_read.setpos()`.

`Wave_write.writeframesraw(data)`

Écrit les trames audio sans corriger `nframes`.

Modifié dans la version 3.4 : N'importe quel *bytes-like object* est maintenant accepté.

`Wave_write.writeframes(data)`

Écrit des trames audio et s'assure que *nframes* est correct. Une erreur est levée si le flux de sortie est non-navigable et si le nombre total de trames écrites après que *data* soit écrit ne correspond pas à la valeur précédemment définie pour *nframes*.

Modifié dans la version 3.4 : N'importe quel *bytes-like object* est maintenant accepté.

Notez qu'il est impossible de définir des paramètres après avoir appelé `writeframes()` ou `writeframesraw()`, et toute tentative en ce sens lève une `wave.Error`.

22.5 chunk — Read IFF chunked data

Source code : [Lib/chunk.py](#)

This module provides an interface for reading files that use EA IFF 85 chunks.¹ This format is used in at least the Audio Interchange File Format (AIFF/AIFF-C) and the Real Media File Format (RMFF). The WAVE audio file format is closely related and can also be read using this module.

A chunk has the following structure :

Offset	Length	Sommaire
0	4	Chunk ID
4	4	Size of chunk in big-endian byte order, not including the header
8	<i>n</i>	Data bytes, where <i>n</i> is the size given in the preceding field
8 + <i>n</i>	0 or 1	Pad byte needed if <i>n</i> is odd and chunk alignment is used

The ID is a 4-byte string which identifies the type of chunk.

The size field (a 32-bit value, encoded using big-endian byte order) gives the size of the chunk data, not including the 8-byte header.

Usually an IFF-type file consists of one or more chunks. The proposed usage of the `Chunk` class defined here is to instantiate an instance at the start of each chunk and read from the instance until it reaches the end, after which a new instance can be instantiated. At the end of the file, creating a new instance will fail with an `EOFError` exception.

class `chunk.Chunk` (*file*, *align=True*, *bigendian=True*, *inclheader=False*)

Class which represents a chunk. The *file* argument is expected to be a file-like object. An instance of this class is specifically allowed. The only method that is needed is `read()`. If the methods `seek()` and `tell()` are present and don't raise an exception, they are also used. If these methods are present and raise an exception, they are expected to not have altered the object. If the optional argument *align* is true, chunks are assumed to be aligned on 2-byte boundaries. If *align* is false, no alignment is assumed. The default value is true. If the optional argument *bigendian* is false, the chunk size is assumed to be in little-endian order. This is needed for WAVE audio files. The default value is true. If the optional argument *inclheader* is true, the size given in the chunk header includes the size of the header. The default value is false.

A `Chunk` object supports the following methods :

getname ()

Returns the name (ID) of the chunk. This is the first 4 bytes of the chunk.

getsize ()

Returns the size of the chunk.

close ()

Close and skip to the end of the chunk. This does not close the underlying file.

1. « EA IFF 85 » Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, January 1985.

The remaining methods will raise `OSError` if called after the `close()` method has been called. Before Python 3.3, they used to raise `IOError`, now an alias of `OSError`.

isatty()

Returns `False`.

seek(*pos*, *whence*=0)

Set the chunk's current position. The *whence* argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file's end). There is no return value. If the underlying file does not allow seek, only forward seeks are allowed.

tell()

Return the current position into the chunk.

read(*size*=-1)

Read at most *size* bytes from the chunk (less if the read hits the end of the chunk before obtaining *size* bytes). If the *size* argument is negative or omitted, read all data until the end of the chunk. An empty bytes object is returned when the end of the chunk is encountered immediately.

skip()

Skip to the end of the chunk. All further calls to `read()` for the chunk will return `b''`. If you are not interested in the contents of the chunk, this method should be called so that the file points to the start of the next chunk.

Notes

22.6 colorsys — Conversions entre les systèmes de couleurs

Code source : <Lib/colorsys.py>

Le module `colorsys` définit les conversions bidirectionnelles des valeurs de couleur entre les couleurs exprimées dans l'espace colorimétrique RVB (Rouge Vert Bleu) utilisé par les écrans d'ordinateur et trois autres systèmes de coordonnées : YIQ, HLS (Hue Lightness Saturation) et HSV (Hue Saturation Value). Les coordonnées dans tous ces espaces colorimétriques sont des valeurs en virgule flottante. Dans l'espace YIQ, la coordonnée Y est comprise entre 0 et 1, mais les coordonnées I et Q peuvent être positives ou négatives. Dans tous les autres espaces, les coordonnées sont toutes comprises entre 0 et 1.

Voir aussi :

Consultez <http://www.poynton.com/ColorFAQ.html> et <https://www.cambridgeincolour.com/tutorials/color-spaces.htm> pour plus d'informations concernant les espaces colorimétriques.

Le module `colorsys` définit les fonctions suivantes :

`colorsys.rgb_to_yiq(r, g, b)`

Convertit la couleur des coordonnées RGB (RVB) vers les coordonnées YIQ.

`colorsys.yiq_to_rgb(y, i, q)`

Convertit la couleur des coordonnées YIQ vers les coordonnées RGB (RVB).

`colorsys.rgb_to_hls(r, g, b)`

Convertit la couleur des coordonnées RGB (RVB) vers les coordonnées HLS (TSV).

`colorsys.hls_to_rgb(h, l, s)`

Convertit la couleur des coordonnées HLS (TSV) vers les coordonnées RGB (RVB).

`colorsys.rgb_to_hsv(r, g, b)`

Convertit la couleur des coordonnées RGB (RVB) vers les coordonnées HSV (TSV).

`colorsys.hsv_to_rgb(h, s, v)`

Convertit la couleur des coordonnées HSV (TSV) vers les coordonnées RGB (RVB).

Exemple :

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

22.7 `imghdr` — Determine the type of an image

Code source : [Lib/imghdr.py](#)

The `imghdr` module determines the type of image contained in a file or byte stream.

The `imghdr` module defines the following function :

`imghdr.what(filename, h=None)`

Tests the image data contained in the file named by *filename*, and returns a string describing the image type. If optional *h* is provided, the *filename* is ignored and *h* is assumed to contain the byte stream to test.

Modifié dans la version 3.6 : Accepte un *path-like object*.

The following image types are recognized, as listed below with the return value from `what()` :

Valeur	Image format
'rgb'	SGI ImgLib Files
'gif'	GIF 87a and 89a Files
'pbm'	Portable Bitmap Files
'pgm'	Portable Graymap Files
'ppm'	Portable Pixmap Files
'tiff'	TIFF Files
'rast'	Sun Raster Files
'xbm'	X Bitmap Files
'jpeg'	JPEG data in JFIF or Exif formats
'bmp'	BMP files
'png'	Portable Network Graphics
'webp'	WebP files
'exr'	OpenEXR Files

Nouveau dans la version 3.5 : The *exr* and *webp* formats were added.

You can extend the list of file types `imghdr` can recognize by appending to this variable :

`imghdr.tests`

A list of functions performing the individual tests. Each function takes two arguments : the byte-stream and an open file-like object. When `what()` is called with a byte-stream, the file-like object will be `None`.

The test function should return a string describing the image type if the test succeeded, or `None` if it failed.

Exemple :

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

22.8 sndhdr — Détermine le type d'un fichier audio

Code source : [Lib/sndhdr.py](#)

Le module `sndhdr` fournit des fonctions permettant d'essayer de déterminer le type de données audio contenues dans un fichier. Lorsque ces fonctions parviennent à déterminer le format de données, elles renvoient un `namedtuple()`, contenant cinq attributs : (`filetype`, `framerate`, `nchannels`, `nframes`, `sampwidth`). La valeur de `type` indique le format de données parmi 'aifc', 'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub', et 'ul'. La valeur de `sampling_rate` sera soit la vraie valeur, soit, si elle est inconnue ou compliquée à obtenir, 0. De même, `channels` vaut soit le nombre de canaux soit 0 s'il ne peut pas être déterminé ou si la valeur est compliquée à décoder. La valeur de `frames` sera soit le nombre de `frames` soit -1. Le dernier élément du tuple, `bits_per_sample` sera soit la taille d'un échantillon en bits, soit 'A' pour A-LAW ou 'U' pour u-LAW.

`sndhdr.what(filename)`

Détermine le type de données audio stockée dans le fichier `filename` en utilisant `whathdr()`. Si elle y parvient, le `namedtuple` décrit plus haut est renvoyé, sinon, `None`.

Modifié dans la version 3.5 : Le type renvoyé passe d'un `tuple` à un `namedtuple`.

`sndhdr.whathdr(filename)`

Détermine le type de données audio contenue dans un fichier, en se basant sur ses entêtes. Le nom du fichier est donné par `filename`. Cette fonction renvoie un `namedtuple` tel que décrit plus haut, si elle y parvient, sinon `None`.

Modifié dans la version 3.5 : Le type renvoyé passe d'un `tuple` à un `namedtuple`.

22.9 ossaudiodev — Access to OSS-compatible audio devices

This module allows you to access the OSS (Open Sound System) audio interface. OSS is available for a wide range of open-source and commercial Unices, and is the standard audio interface for Linux and recent versions of FreeBSD.

Modifié dans la version 3.3 : Operations in this module now raise `OSError` where `IOError` was raised.

Voir aussi :

Open Sound System Programmer's Guide the official documentation for the OSS C API

The module defines a large number of constants supplied by the OSS device driver; see `<sys/soundcard.h>` on either Linux or FreeBSD for a listing.

`ossaudiodev` defines the following variables and functions :

exception `ossaudiodev.OSSAudioError`

This exception is raised on certain errors. The argument is a string describing what went wrong.

(If `ossaudiodev` receives an error from a system call such as `open()`, `write()`, or `ioctl()`, it raises `OSError`. Errors detected directly by `ossaudiodev` result in `OSSAudioError`.)

(For backwards compatibility, the exception class is also available as `ossaudiodev.error`.)

`ossaudiodev.open(mode)`

`ossaudiodev.open(device, mode)`

Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as `read()`, `write()`, and `fileno()` (although there are subtle differences between conventional Unix read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

device is the audio device filename to use. If it is not specified, this module first looks in the environment variable `AUDIODEV` for a device to use. If not found, it falls back to `/dev/dsp`.

mode is one of `'r'` for read-only (record) access, `'w'` for write-only (playback) access and `'rw'` for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex : they can be opened for reading or writing, but not both at once.

Note the unusual calling syntax : the *first* argument is optional, and the second is required. This is a historical artifact for compatibility with the older `linuxaudiodev` module which *ossaudiodev* supersedes.

`ossaudiodev.openmixer([device])`

Open a mixer device and return an OSS mixer device object. *device* is the mixer device filename to use. If it is not specified, this module first looks in the environment variable `MIXERDEV` for a device to use. If not found, it falls back to `/dev/mixer`.

22.9.1 Audio Device Objects

Before you can write to or read from an audio device, you must call three methods in the correct order :

1. `setfmt()` to set the output format
2. `channels()` to set the number of channels
3. `speed()` to set the sample rate

Alternately, you can use the `setparameters()` method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by *open()* define the following methods and (read-only) attributes :

`oss_audio_device.close()`

Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

`oss_audio_device.fileno()`

Return the file descriptor associated with the device.

`oss_audio_device.read(size)`

Read *size* bytes from the audio input and return them as a Python string. Unlike most Unix device drivers, OSS audio devices in blocking mode (the default) will block *read()* until the entire requested amount of data is available.

`oss_audio_device.write(data)`

Write a *bytes-like object* *data* to the audio device and return the number of bytes written. If the audio device is in blocking mode (the default), the entire data is always written (again, this is different from usual Unix device semantics). If the device is in non-blocking mode, some data may not be written—see *writeall()*.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

`oss_audio_device.writeall(data)`

Write a *bytes-like object* *data* to the audio device : waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until *data* has been completely written. If the device is in blocking mode (the default), this has the same effect as *write()* ; *writeall()* is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

Modifié dans la version 3.2 : Audio device objects also support the context management protocol, i.e. they can be used in a `with` statement.

The following methods each map to exactly one `ioctl()` system call. The correspondence is obvious : for example, `setfmt()` corresponds to the `SNDCTL_DSP_SETFMT` `ioctl`, and `sync()` to `SNDCTL_DSP_SYNC` (this can be useful when consulting the OSS documentation). If the underlying `ioctl()` fails, they all raise `OSError`.

`oss_audio_device.nonblock()`

Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

`oss_audio_device.getfmts()`

Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are :

Format	Description
AFMT_MU_LAW	a logarithmic encoding (used by Sun .au files and /dev/audio)
AFMT_A_LAW	a logarithmic encoding
AFMT_IMA_ADPCM	a 4:1 compressed format defined by the Interactive Multimedia Association
AFMT_U8	Unsigned, 8-bit audio
AFMT_S16_LE	Signed, 16-bit audio, little-endian byte order (as used by Intel processors)
AFMT_S16_BE	Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)
AFMT_S8	Signed, 8 bit audio
AFMT_U16_LE	Unsigned, 16-bit little-endian audio
AFMT_U16_BE	Unsigned, 16-bit big-endian audio

Consult the OSS documentation for a full list of audio formats, and note that most devices support only a subset of these formats. Some older devices only support `AFMT_U8` ; the most common format used today is `AFMT_S16_LE`.

`oss_audio_device.setfmt(format)`

Try to set the current audio format to *format*—see `getfmts()` for a list. Returns the audio format that the device was set to, which may not be the requested format. May also be used to return the current audio format—do this by passing an « audio format » of `AFMT_QUERY`.

`oss_audio_device.channels(nchannels)`

Set the number of output channels to *nchannels*. A value of 1 indicates monophonic sound, 2 stereophonic. Some devices may have more than 2 channels, and some high-end devices may not support mono. Returns the number of channels the device was set to.

`oss_audio_device.speed(samplerate)`

Try to set the audio sampling rate to *samplerate* samples per second. Returns the rate actually set. Most sound devices don't support arbitrary sampling rates. Common rates are :

Rate	Description
8000	default rate for /dev/audio
11025	speech recording
22050	
44100	CD quality audio (at 16 bits/sample and 2 channels)
96000	DVD quality audio (at 24 bits/sample)

`oss_audio_device.sync()`

Wait until the sound device has played every byte in its buffer. (This happens implicitly when the device is closed.) The OSS documentation recommends closing and re-opening the device rather than using `sync()`.

`oss_audio_device.reset()`

Immediately stop playing or recording and return the device to a state where it can accept commands. The OSS documentation recommends closing and re-opening the device after calling `reset()`.

`oss_audio_device.post()`

Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several `ioctl`s, or one `ioctl` and some simple calculations.

`oss_audio_device.setparameters(format, nchannels, samplerate[, strict=False])`

Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. *format*, *nchannels*, and *samplerate* should be as specified in the `setfmt()`, `channels()`, and `speed()` methods. If *strict* is true, `setparameters()` checks to see if each parameter was actually set to the requested value, and raises `OSSAudioError` if not. Returns a tuple (*format*, *nchannels*, *samplerate*) indicating the parameter values that were actually set by the device driver (i.e., the same as the return values of `setfmt()`, `channels()`, and `speed()`).

For example,

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

is equivalent to

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

`oss_audio_device.bufsize()`

Returns the size of the hardware buffer, in samples.

`oss_audio_device.obufcount()`

Returns the number of samples that are in the hardware buffer yet to be played.

`oss_audio_device.obuffree()`

Returns the number of samples that could be queued into the hardware buffer to be played without blocking.

Audio device objects also support several read-only attributes :

`oss_audio_device.closed`

Boolean indicating whether the device has been closed.

`oss_audio_device.name`

String containing the name of the device file.

`oss_audio_device.mode`

The I/O mode for the file, either "r", "rw", or "w".

22.9.2 Mixer Device Objects

The mixer object provides two file-like methods :

`oss_mixer_device.close()`

This method closes the open mixer device file. Any further attempts to use the mixer after this file is closed will raise an `OSError`.

`oss_mixer_device.fileno()`

Returns the file handle number of the open mixer device file.

Modifié dans la version 3.2 : Mixer objects also support the context management protocol.

The remaining methods are specific to audio mixing :

`oss_mixer_device.controls()`

This method returns a bitmask specifying the available mixer controls (« Control » being a specific mixable « channel », such as `SOUND_MIXER_PCM` or `SOUND_MIXER_SYNTH`). This bitmask indicates a subset of all available mixer controls—the `SOUND_MIXER_*` constants defined at module level. To determine if, for example, the current mixer object supports a PCM mixer, use the following Python code :

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

For most purposes, the `SOUND_MIXER_VOLUME` (master volume) and `SOUND_MIXER_PCM` controls should suffice—but code that uses the mixer should be flexible when it comes to choosing mixer controls. On the Gravis Ultrasound, for example, `SOUND_MIXER_VOLUME` does not exist.

`oss_mixer_device.stereocontrols()`

Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo ; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with `controls()` to determine which).

See the code example for the `controls()` function for an example of getting data from a bitmask.

`oss_mixer_device.recontrols()`

Returns a bitmask specifying the mixer controls that may be used to record. See the code example for `controls()` for an example of reading from a bitmask.

`oss_mixer_device.get(control)`

Returns the volume of a given mixer control. The returned volume is a 2-tuple (`left_volume`, `right_volume`). Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

Raises `OSSAudioError` if an invalid control is specified, or `OSError` if an unsupported control is specified.

`oss_mixer_device.set(control, (left, right))`

Sets the volume for a given mixer control to (`left`, `right`). `left` and `right` must be ints and between 0 (silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple. Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard's mixers.

Raises `OSSAudioError` if an invalid mixer control was specified, or if the specified volumes were out-of-range.

`oss_mixer_device.get_recsrc()`

This method returns a bitmask indicating which control(s) are currently being used as a recording source.

`oss_mixer_device.set_recsrc(bitmask)`

Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources) if successful ; raises `OSError` if an invalid source was specified. To set the current recording source to the microphone input :

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```


Les modules décrits dans ce chapitre vous aident à rédiger des programmes indépendants des langues et des cultures en fournissant des mécanismes pour sélectionner une langue à utiliser dans les messages, ou en adaptant la sortie aux conventions culturelles.

La liste des modules documentés dans ce chapitre est :

23.1 `gettext` — Services d'internationalisation multilingue

Code source : [Lib/gettext.py](#)

Le module `gettext` fournit un service d'internationalisation (*I18N*) et de localisation linguistique (*L10N*) pour vos modules et applications Python. Il est compatible avec l'API du catalogue de messages GNU `gettext` et à un plus haut niveau, avec l'API basée sur les classes qui serait peut-être plus adaptée aux fichiers Python. L'interface décrite ci-dessous vous permet d'écrire les textes de vos modules et applications dans une langue naturelle, puis de fournir un catalogue de traductions pour les lancer ensuite dans d'autres langues naturelles.

Quelques astuces sur la localisation de vos modules et applications Python sont également données.

23.1.1 API GNU `gettext`

Le module `gettext` définit l'API suivante, qui est très proche de l'API de GNU `gettext`. Si vous utilisez cette API, cela affectera la traduction de toute votre application. C'est souvent le comportement attendu si votre application est monolingue, avec le choix de la langue qui dépend des paramètres linguistiques de l'utilisateur. Si vous localisez un module Python ou si votre application a besoin de changer de langue à la volée, il est plus judicieux d'utiliser l'API basée sur des classes.

`gettext.bindtextdomain` (*domain*, *localedir*=None)

Lie *domain* au répertoire *localedir* des localisations. Plus spécifiquement, `gettext` va chercher les fichiers binaires `.mo` pour un domaine donné, en utilisant le chemin suivant (sous Unix) : `localedir/language/`

`LC_MESSAGES/domain.mo`, où *language* est contenu respectivement dans l'une des variables d'environnement suivantes : `LANGUAGE`, `LC_ALL`, `LC_MESSAGES` et `LANG`.

Si *localedir* n'est pas renseigné ou vaut `None`, alors le lien actuel de *domain* est renvoyé.¹

`gettext.bind_textdomain_codeset (domain, codeset=None)`

Lie *domain* à *codeset*, en changeant l'encodage des chaînes d'octets retournées par les fonctions `lgettext()`, `ldgettext()`, `lnggettext()` et `ldngettext()`. Si *codeset* n'est pas renseigné, alors le lien actuel est renvoyé.

`gettext.textdomain (domain=None)`

Change ou interroge le domaine global actuel. Si *domain* vaut `None`, alors le domaine global actuel est renvoyé. Sinon, le domaine global est positionné à *domain*, puis renvoyé.

`gettext.gettext (message)`

Renvoie la traduction localisée de *message*, en se basant sur le domaine global actuel, la langue et le répertoire des localisations. Cette fonction est typiquement renommée `_()` dans le namespace courant (voir les exemples ci-dessous).

`gettext.dgettext (domain, message)`

Comme `gettext()`, mais cherche le message dans le domaine spécifié.

`gettext.ngettext (singular, plural, n)`

Comme `gettext()`, mais prend en compte les formes au pluriel. Si une traduction a été trouvée, utilise la formule pour trouver le pluriel à *n* et renvoie le message généré (quelques langues ont plus de deux formes au pluriel). Si aucune traduction n'a été trouvée, renvoie *singular* si *n* vaut 1, *plural* sinon.

La formule pour trouver le pluriel est récupérée dans l'entête du catalogue. C'est une expression en C ou en Python qui a une variable libre *n* et qui évalue l'index du pluriel dans le catalogue. Voir la [documentation de GNU gettext](#) pour la syntaxe précise à utiliser dans les fichiers `.po` et pour les formules dans différents langues.

`gettext.dngettext (domain, singular, plural, n)`

Comme `ngettext()`, mais cherche le message dans le domaine spécifié.

`gettext.lgettext (message)`

`gettext.ldgettext (domain, message)`

`gettext.lnggettext (singular, plural, n)`

`gettext.ldngettext (domain, singular, plural, n)`

Équivalent aux fonctions correspondantes non préfixées par `l` (`gettext()`, `dgettext()`, `ngettext()` et `dngettext()`), mais la traduction est retournée en tant que chaîne d'octets, encodée avec l'encodage du système si aucun autre n'a été explicitement défini avec `bind_textdomain_codeset()`.

Avertissement : Ces fonctions sont à éviter en Python 3 car elles renvoient des octets encodés. Il est préférable d'utiliser des alternatives qui renvoient de l'Unicode, puisque beaucoup d'applications Python voudront manipuler du texte lisible par des humains plutôt que des octets. En outre, il est possible que vous obteniez des exceptions non prévues liées à Unicode s'il y a des soucis d'encodage avec les chaînes de caractères traduites. Il est d'ailleurs probable que les fonctions `l*` deviennent obsolètes dans les versions futures de Python à cause de leurs problèmes et limitations inhérents.

Notez que GNU **gettext** a aussi une méthode `dcgettext()`, mais elle a été considérée comme inutile et donc actuellement marquée comme non implémentée.

Voici un exemple classique d'utilisation de cette API :

1. Le dossier par défaut pour les localisations dépend du système : par exemple, sur RedHat c'est `/usr/share/locale`, mais sur Solaris c'est `/usr/lib/locale`. Le module `gettext` n'essaie pas de prendre en charge ce paramètre système, mais utilise le dossier `sys.prefix/share/locale`. C'est pour cette raison qu'il est toujours préférable d'appeler `bindtextdomain()` en donnant explicitement un chemin absolu au début de votre application.

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

23.1.2 API basée sur les classes

L'API du module `gettext` basée sur les classes vous donne plus de flexibilité et est plus pratique que l'API de GNU `gettext`. Son utilisation est recommandée pour localiser vos applications et modules Python. `gettext` définit une classe « translations » qui analyse syntaxiquement les fichiers au format GNU `.mo`, et qui possède des méthodes pour renvoyer des chaînes de caractères. Les instances de cette classe « translations » peuvent également s'installer dans l'espace de noms natif en tant que fonction `_()`.

`gettext.find(domain, localedir=None, languages=None, all=False)`

Cette fonction implémente l'algorithme standard de recherche de fichier `.mo`. Il prend en entrée un *domain*, tout comme la fonction `textdomain()`. Le paramètre optionnel *localedir* est le même que celui de `bindtextdomain()`. Le paramètre optionnel *languages* est une liste de chaînes de caractères correspondants au code d'une langue.

Si *localedir* n'est pas renseigné, alors le répertoire de la locale par défaut du système est utilisé.² Si *languages* n'est pas renseigné, alors les variables d'environnement suivantes sont utilisées : `LANGUAGE`, `LC_ALL`, `LC_MESSAGES` et `LANG`. La première à renvoyer une valeur non vide est alors utilisée pour *languages*. Ces variables d'environnement doivent contenir une liste de langues, séparées par des deux-points, qui sera utilisée pour générer la liste des codes de langues attendue.

Recherche avec `find()`, découvre et normalise les langues, puis itère sur la liste obtenue afin de trouver un fichier de traduction existant et correspondant :

`localedir/language/LC_MESSAGES/domain.mo`

Le premier nom de fichier trouvé est renvoyé par `find()`. Si aucun fichier n'a été trouvé, alors `None` est renvoyé. Si *all* est vrai, est renvoyée la liste de tous les noms de fichiers, dans l'ordre dans lequel ils apparaissent dans *languages* ou dans les variables d'environnement.

`gettext.translation(domain, localedir=None, languages=None, class_=None, fallback=False, codeset=None)`

Renvoie une instance de la classe `Translations` en se basant sur *domain*, *localedir* et *languages*, qui sont d'abord passés en argument de `find()` afin d'obtenir une liste de chemin des fichiers `.mo` associés. Les instances avec des noms de fichiers `.mo` identiques sont mises en cache. La classe réellement instanciée est soit *class_* si renseigné, soit une classe `GNUTranslations`. Le constructeur de cette classe doit prendre en argument un seul *file object*. Si renseigné, *codeset* modifiera le jeu de caractères utilisé pour encoder les chaînes de caractères traduites, dans les méthodes `gettext()` et `gettext()`.

Si plusieurs fichiers ont été trouvés, les derniers sont utilisés comme substitut des premiers. Pour rendre possible cette substitution, `copy.copy()` est utilisé pour copier chaque objet traduit depuis le cache ; les vraies données de l'instance étant toujours recopiées dans le cache.

Si aucun fichier `.mo` n'a été trouvé, soit *fallback* vaut `False` (valeur par défaut) et une exception `OSError` est levée, soit *fallback* vaut `True` et une instance `NullTranslations` est renvoyée.

Modifié dans la version 3.3 : Avant, c'était l'exception `IOError` qui était levée, au lieu de `OSError`.

`gettext.install(domain, localedir=None, codeset=None, names=None)`

Positionne la fonction `_()` dans l'espace de nommage natif de Python, en se basant sur *domain*, *localedir* et *codeset*, qui sont passés en argument de la fonction `translation()`.

Concernant le paramètre *names*, se référer à la description de la méthode `install()`.

Habituellement, la fonction `_()` est appliquée aux chaînes de caractères qui doivent être traduites comme suit :

2. Voir la note de `bindtextdomain()` ci-dessus.

```
print(_('This string will be translated.'))
```

Pour plus de confort, il vaut mieux positionner la fonction `_()` dans l'espace de nommage natif de Python pour la rendre plus accessible dans tous les modules de votre application.

La classe `NullTranslations`

Les classes de traduction implémentent le fait de passer d'une chaîne de caractères du fichier original à traduire à la traduction de celle-ci. La classe de base utilisée est `NullTranslations`. C'est l'interface de base à utiliser lorsque vous souhaitez écrire vos propres classes spécifiques à la traduction. Voici les méthodes de `NullTranslations` :

class `gettext.NullTranslations` (*fp=None*)

Prend un paramètre optionnel un *file object* *fp*, qui est ignoré par la classe de base. Initialise les variables d'instance « protégées » `_info` et `_charset`, définies par des classes dérivées, tout comme `_fallback` qui est définie au travers de `add_fallback()`. Puis appelle `self._parse(fp)` si *fp* ne vaut pas `None`.

_parse (*fp*)

Cette méthode, non exécutée dans la classe de base, prend en paramètre un objet fichier *fp* et lit les données de ce dernier. Si vous avez un catalogue de messages dont le format n'est pas pris en charge, vous devriez surcharger cette méthode pour analyser votre format.

add_fallback (*fallback*)

Ajoute *fallback* comme objet de substitution pour l'objet de traduction courant. Un objet de traduction devrait interroger cet objet de substitution s'il ne peut fournir une traduction pour un message donné.

gettext (*message*)

Si un objet de substitution a été défini, transmet `gettext()` à celui-ci. Sinon, renvoie *message*. Surchargé dans les classes dérivées.

ngettext (*singular, plural, n*)

Si un objet de substitution a été défini, transmet `ngettext()` à celui-ci. Sinon, renvoie *singular* si *n* vaut 1, *plural* sinon. Surchargé dans les classes dérivées.

lgettext (*message*)

lngettext (*singular, plural, n*)

Équivalent de `gettext()` et `ngettext()`, mais la traduction est renvoyée sous la forme d'une chaîne d'octets, encodée avec l'encodage du système si aucun autre n'a été défini avec `set_output_charset()`. Surchargé dans les classes dérivées.

Avertissement : L'utilisation de ces méthodes doivent être évitée en Python 3. Voir l'avertissement de la fonction `lgettext()`.

info ()

Renvoie la variable « protégée » `_info`.

charset ()

Renvoie l'encodage du fichier du catalogue de messages.

output_charset ()

Renvoie l'encodage utilisé par `lgettext()` et `lngettext()` pour la traduction des messages.

set_output_charset (*charset*)

Modifie l'encodage utilisé pour la traduction des messages.

install (*names=None*)

Cette méthode positionne `gettext()` dans l'espace de nommage natif, en le liant à `_`.

Si le paramètre *names* est renseigné, celui-ci doit être une séquence contenant les noms des fonctions que vous souhaitez positionner dans l'espace de nommage natif, en plus de `_()`. Les noms pris en charge sont 'gettext', 'ngettext', 'lgettext' et 'lngettext'.

Notez que ce n'est là qu'un moyen parmi d'autres, quoique le plus pratique, pour rendre la fonction `_()` accessible à votre application. Puisque cela affecte toute l'application, et plus particulièrement l'espace de

nommage natif, les modules localisés ne devraient jamais y positionner `_()`. Au lieu de cela, ces derniers doivent plutôt utiliser le code suivant pour rendre `_()` accessible par leurs modules :

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

Cela met `_()` dans l'espace de nommage global du module uniquement et donc n'affectera ses appels que dans ce module.

La classe GNUTranslations

Le module `gettext` fournit une classe supplémentaire qui hérite de `NullTranslations` : `GNUTranslations`. Cette classe surcharge `_parse()` pour permettre de lire les fichiers GNU `gettext` `.mo` au format petit et gros-boutiste.

`GNUTranslations` analyse les métadonnées optionnelles du catalogue de traduction. Il est d'usage avec GNU `gettext` d'utiliser une métadonnée pour traduire la chaîne vide. Cette métadonnée est un ensemble de paires de la forme `clef: valeur` comme définie par la [RFC 822](#), et doit contenir la clef `Project-Id-Version`. Si la clef `Content-Type` est trouvée dans une métadonnée, alors la propriété `charset` (jeu de caractères) est utilisée pour initialiser la variable d'instance « protégée » `_charset`, sinon cette dernière est positionnée à `None`. Si l'encodage du jeu de caractères est spécifié, tous les messages (identifiants et chaînes de caractères) lus depuis le catalogue sont convertis en chaînes Unicode via cet encodage, ou via l'encodage ASCII si non renseigné.

Et puisque les identifiants des messages sont également lus comme des chaînes Unicode, toutes les méthodes `*gettext()` les considéreront ainsi, et pas comme des chaînes d'octets.

La totalité des paires `clef / valeur` est insérée dans un dictionnaire et représente la variable d'instance « protégée » `_info`.

Si le nombre magique du fichier `.mo` est invalide, le numéro de la version majeure inattendu, ou si d'autres problèmes apparaissent durant la lecture du fichier, instancier une classe `GNUTranslations` peut lever une exception `OSError`.

class gettext.GNUTranslations

Les méthodes suivantes, provenant de l'implémentation de la classe de base, ont été surchargées :

gettext(*message*)

Recherche l'identifiant de *message* dans le catalogue et renvoie le message de la chaîne de caractères correspondante comme une chaîne Unicode. Si aucun identifiant n'a été trouvé pour *message* et qu'un substitut a été défini, la recherche est transmise à la méthode `gettext()` du substitut. Sinon, l'identifiant de *message* est renvoyé.

ngettext(*singular*, *plural*, *n*)

Effectue une recherche sur les formes plurielles de l'identifiant d'un message. *singular* est utilisé pour la recherche de l'identifiant dans le catalogue, alors que *n* permet de savoir quelle forme plurielle utiliser. La chaîne de caractère du message renvoyée est une chaîne Unicode.

Si l'identifiant du message n'est pas trouvé dans le catalogue et qu'un substitut a été spécifié, la requête est transmise à la méthode `ngettext()` du substitut. Sinon, est renvoyé *singular* lorsque *n* vaut 1, *plural* dans tous les autres cas.

Voici un exemple :

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

lgettext(*message*)

lgettext (*singular, plural, n*)

Équivalent de `gettext()` et `ngettext()`, mais la traduction est renvoyée sous la forme d'une chaîne d'octets, encodée avec l'encodage du système si aucun autre n'a été défini avec `set_output_charset()`.

Avertissement : L'utilisation de ces méthodes doivent être évitée en Python 3. Voir l'avertissement de la fonction `lgettext()`.

Support du catalogue de message de Solaris

Le système d'exploitation Solaris possède son propre format de fichier binaire `.mo`, mais pour l'heure, puisqu'on ne peut trouver de documentation sur ce format, il n'est pas géré.

Le constructeur *Catalog*

GNOME utilise une version du module `gettext` de James Henstridge, mais qui a une API légèrement différente. D'après la documentation, elle s'utilise ainsi :

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

Pour des raisons de compatibilité avec cet ancien module, la fonction `Catalog()` est un alias de la fonction `translation()` décrite ci-dessous.

Une différence entre ce module et celui de Henstridge : les objets de son catalogue étaient accessibles depuis un schéma de l'API, mais cela semblait ne pas être utilisé et donc n'est pas pris en charge.

23.1.3 Internationaliser vos programmes et modules

L'internationalisation (*I18N*) consiste à permettre à un programme de recevoir des traductions dans plusieurs langues. La localisation (*L10N*) consiste à adapter un programme à la langue et aux habitudes culturelles locales, une fois celui-ci internationalisé. Afin de fournir du texte multilingue à votre programme Python, les étapes suivantes sont nécessaires :

1. préparer votre programme ou module en marquant spécifiquement les chaînes à traduire
2. lancer une suite d'outils sur les fichiers contenant des chaînes à traduire pour générer des catalogues de messages brut
3. créer les traductions spécifiques à une langue des catalogues de messages
4. utiliser le module `gettext` pour que les chaînes de caractères soient bien traduites

Afin de préparer votre code à être traduit (*I18N*), vous devrez rechercher toutes les chaînes de caractères de vos fichiers. À chaque chaîne de caractères à traduire doit être appliqué le marqueur `_('...')` — c'est-à-dire en appelant la fonction `_()`. Par exemple :

```
filename = 'mylog.txt'
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

Dans cet exemple, la chaîne `'writing a log message'` est maquée comme traduite, contrairement aux chaînes `'mylog.txt'` et `'w'`.

Il existe quelques outils pour extraire les chaînes de caractères destinées à la traduction. Le programme d'origine GNU **gettext** ne prenait en charge que les codes sources en C ou C++, mais sa version étendue **xgettext** peut lire du code écrit dans de nombreux langages, dont le Python, afin de trouver les chaînes notées comme traduisibles. **Babel** est une bibliothèque en Python d'internationalisation, qui inclut un script `pybabel` permettant d'extraire et de compiler des catalogues de messages. Le programme de François Pinard, nommé **xpot**, fait de même et est disponible dans son [paquet po-utils](#).

(Python inclut également des versions en Python de ces programmes, `pygettext.py` et `msgfmt.py`, que certaines distributions Python installeront pour vous. `pygettext.py` est similaire à **xgettext**, mais ne comprend que le code source écrit en Python et ne peut prendre en charge d'autres langages de programmation tels que le C ou C++. `pygettext.py` possède une interface en ligne de commande similaire à celle de **xgettext** — pour plus de détails sur son utilisation, exécuter `pygettext.py --help`. `msgfmt.py` est compatible avec GNU **msgfmt**. Avec ces deux programmes, vous ne devriez pas avoir besoin du paquet GNU **gettext** pour internationaliser vos applications en Python.)

xgettext, **pygettext** et d'autres outils similaires génèrent des fichiers `.po` représentant les catalogues de messages. Il s'agit de fichiers structurés et lisibles par un être humain, qui contiennent toutes les chaînes du code source marquées comme traduisible, ainsi que leur traduction à utiliser.

Les copies de ces fichiers `.po` sont ensuite remises à des êtres humains qui traduisent le contenu pour chaque langue naturelle prise en charge. Pour chacune des langues, ces derniers renvoient la version complétée sous la forme d'un fichier `<code-langue>.po` qui a été compilé dans un fichier binaire `.mo` représentant le catalogue lisible par une machine à l'aide du programme **msgfmt**. Les fichiers `.mo` sont utilisés par le module `gettext` pour la traduction lors de l'exécution.

La façon dont vous utilisez le module `gettext` dans votre code dépend de si vous internationalisez un seul module ou l'ensemble de votre application. Les deux sections suivantes traitent chacune des cas.

Localiser votre module

Si vous localisez votre module, veillez à ne pas faire de changements globaux, e.g. dans l'espace de nom natif. Vous ne devriez pas utiliser l'API GNU **gettext** mais plutôt celle basée sur les classes.

Disons que votre module s'appelle « spam » et que les fichiers `.mo` de traduction dans les différentes langues naturelles soient dans `/usr/share/locale` au format GNU **gettext**. Voici ce que vous pouvez alors mettre en haut de votre module :

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

Localiser votre application

Si vous localisez votre application, vous pouvez positionner la fonction `_()` de manière globale dans l'espace de nommage natif, généralement dans le fichier principal de votre application. Cela permettra à tous les fichiers de votre application de n'utiliser que `_('...')` sans devoir le redéfinir explicitement dans chaque fichier.

Dans ce cas, vous n'aurez à ajouter que le bout de code suivant au fichier principal de votre application :

```
import gettext
gettext.install('myapplication')
```

Si vous avez besoin de définir le dossier des localisations, vous pouvez le mettre en argument de la fonction `install()` :

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

Changer de langue à la volée

Si votre programme a besoin de prendre en charge plusieurs langues en même temps, vous pouvez créer plusieurs instances de traduction, puis basculer entre elles de façon explicite, comme ceci :

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Traductions différées

Dans la plupart des cas, en programmation, les chaînes de caractères sont traduites à l'endroit où on les écrit. Cependant, il peut arriver que vous ayez besoin de traduire une chaîne de caractères un peu plus loin. Un exemple classique est :

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]

# ...
for a in animals:
    print(a)
```

Ici, vous voulez marquer les chaînes de caractères de la liste `animals` comme étant traduisibles, mais ne les traduire qu'au moment de les afficher.

Voici un moyen de gérer ce cas :

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

Cela fonctionne car la définition factice de `_()` renvoie simplement la chaîne de caractères passée en entrée. Et cette définition factice va temporairement outrepasser toute autre définition de `_()` dans l'espace de nommage natif (jusqu'à l'utilisation de la commande `del`). Attention toutefois si vous avez déjà une autre définition de `_()` dans l'espace de nommage local.

À noter qu'à la deuxième utilisation de `_()`, « a » ne sera pas vue comme traduisible par le programme `gettext` car ce n'est pas un chaîne au sens propre.

Voici une autre solution :

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

Dans ce cas, les chaînes à traduire sont identifiées avec la fonction `N_()`, qui n'entre pas en conflit avec définition de `_()`. Cependant, il faudra apprendre à votre programme d'extraction de messages à rechercher les chaînes de caractères à traduire parmi celles ayant le marqueur `N_()`. `xgettext`, `pygettext`, `pybabel extract` et `xpot` prennent tous en charge cela grâce à l'option en ligne de commande `-k`. Le choix du nom `N_()` ici est totalement arbitraire et aurait très bien pu être `MarqueurDeTraduction()`.

23.1.4 Remerciements

Les personnes suivantes ont contribué au code, ont fait des retours, ont participé aux suggestions de conception et aux implémentations précédentes, et ont partagé leur expérience précieuse pour la création de ce module :

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

Notes

23.2 `locale` — Services d'internationalisation

Code source : [Lib/locale.py](#)

Le module `locale` donne accès à la base de données et aux fonctionnalités des paramètres linguistiques définis par POSIX. Le mécanisme des paramètres linguistiques de POSIX permet aux développeurs de faire face à certaines problématiques culturelles dans une application, sans avoir à connaître toutes les spécificités de chaque pays où le logiciel est exécuté.

Le module `locale` est implémenté au-dessus du module `_locale`, qui lui-même utilise l'implémentation du paramètre régional ANSI C si disponible.

Le module `locale` définit l'exception et les fonctions suivantes :

exception `locale.Error`

Exception levée lorsque le paramètre régional passé en paramètre de `setlocale()` n'est pas reconnu.

`locale.setlocale(category, locale=None)`

Si `locale` ne vaut pas `None`, `setlocale()` modifie le paramètre régional pour la catégorie `category`. Les catégories disponibles sont listées dans la description des données ci-dessous. `locale` peut être une chaîne de caractères ou un itérable de deux chaînes de caractères (code de la langue et encodage). Si c'est un itérable, il est converti en un nom de paramètre régional à l'aide du moteur d'aliasing fait pour. Si c'est une chaîne vide, les paramètres par défaut de l'utilisateur sont utilisés. Si la modification du paramètre régional échoue, l'exception `Error` est levée. Si elle fonctionne, le nouveau paramètre est renvoyé.

Si `locale` est omis ou vaut `None`, le paramètre actuel de `category` est renvoyé.

`setlocale()` n'est pas *thread-safe* sur la plupart des systèmes. Les applications commencent généralement par un appel de :

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

Cela définit les paramètres régionaux dans toutes les catégories sur ceux par défaut de l'utilisateur (habituellement spécifiés dans la variable d'environnement `LANG`). Si les paramètres régionaux ne sont pas modifiés par la suite, l'utilisation de fils d'exécution ne devrait pas poser de problèmes.

`locale.localeconv()`

Renvoie la base de données des conventions locales sous forme de dictionnaire. Ce dictionnaire a les chaînes de caractères suivantes comme clés :

Catégorie	Clé	Signification
<i>LC_NUMERIC</i>	'decimal_point'	Caractère du séparateur décimal (entre la partie entière et la partie décimale).
	'grouping'	Séquence de nombres spécifiant les positions relatives attendues pour 'thousands_sep' (séparateur de milliers). Si la séquence se termine par <i>CHAR_MAX</i> , aucun autre regroupement n'est effectué. Si la séquence se termine par un 0, la dernière taille du groupe est utilisée à plusieurs reprises.
	'thousands_sep'	Caractère utilisé entre les groupes (séparateur de milliers).
<i>LC_MONETARY</i>	'int_curr_symbol'	Symbole monétaire international.
	'currency_symbol'	Symbole monétaire local.
	'p_cs_precedes/n_cs_precedes'	Si le symbole monétaire précède ou non la valeur (pour les valeurs positives, resp. négatives).
	'p_sep_by_space/n_sep_by_space'	Si le symbole monétaire est séparé de la valeur par une espace ou non (pour les valeurs positives, resp. négatives).
	'mon_decimal_point'	Séparateur décimal (entre la partie entière et la partie décimale) utilisé pour les valeurs monétaires.
	'frac_digits'	Nombre de décimales utilisées dans le format local des valeurs monétaires.
	'int_frac_digits'	Nombre de décimales utilisées dans le format international des valeurs monétaires.
	'mon_thousands_sep'	Séparateur de groupe utilisé pour les valeurs monétaires.
	'mon_grouping'	Équivalent de 'grouping', utilisé pour les valeurs monétaires.
	'positive_sign'	Symbole utilisé pour indiquer qu'une valeur monétaire est positive.
	'negative_sign'	Symbole utilisé pour indiquer qu'une valeur monétaire est négative.
	'p_sign_posn/n_sign_posn'	Position du signe (pour les valeurs positives, resp. négatives), voir ci-dessous.

Toutes les valeurs numériques peuvent être définies à *CHAR_MAX* pour indiquer qu'il n'y a pas de valeur spécifiée pour ces paramètres régionaux.

Les valeurs possibles pour 'p_sign_posn' et 'n_sign_posn' sont données ci-dessous.

Valeur	Explication
0	Le symbole monétaire et la valeur sont entourés de parenthèses.
1	Le signe doit précéder la valeur et le symbole monétaire.
2	Le signe doit suivre la valeur et le symbole monétaire.
3	Le signe doit précéder immédiatement la valeur.
4	Le signe doit suivre immédiatement la valeur.
CHAR_MAX	Rien n'est spécifié dans ces paramètres régionaux.

The function sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale or the `LC_MONETARY` locale if locales are different and numeric or monetary strings are non-ASCII. This temporary change affects other threads. Modifié dans la version 3.6.5 : La fonction définit maintenant la valeur du paramètre `LC_CTYPE` à celle du paramètre `LC_NUMERIC` temporairement dans certains cas.

`locale.nl_langinfo(option)`

Renvoie quelques informations spécifiques aux paramètres régionaux sous forme de chaîne. Cette fonction n'est pas disponible sur tous les systèmes et l'ensemble des options possibles peut également varier d'une plateforme à l'autre. Les valeurs possibles pour les arguments sont des nombres, pour lesquels des constantes symboliques sont disponibles dans le module *locale*.

La fonction `nl_langinfo()` accepte l'une des clés suivantes. La plupart des descriptions sont extraites des descriptions correspondantes dans la bibliothèque GNU C.

`locale.CODESET`

Récupère une chaîne avec le nom de l'encodage des caractères utilisé par le paramètre régional sélectionné.

`locale.D_T_FMT`

Récupère une chaîne qui peut être utilisée comme une chaîne de format par `time.strftime()` afin de représenter la date et l'heure pour un paramètre régional spécifique.

`locale.D_FMT`

Récupère une chaîne qui peut être utilisée comme une chaîne de format par `time.strftime()` afin de représenter une date pour un paramètre régional spécifique.

`locale.T_FMT`

Récupère une chaîne qui peut être utilisée comme une chaîne de format par `time.strftime()` afin de représenter une heure pour un paramètre régional spécifique.

`locale.T_FMT_AMPM`

Récupère une chaîne de format pour `time.strftime()` afin de représenter l'heure au format am / pm.

`DAY_1 ... DAY_7`

Récupère le nom du n-ième jour de la semaine.

Note : Cela suit la convention américaine qui définit `DAY_1` comme étant dimanche, et non la convention internationale (ISO 8601) où lundi est le premier jour de la semaine.

`ABDAY_1 ... ABDAY_7`

Récupère l'abréviation du n-ième jour de la semaine.

`MON_1 ... MON_12`

Récupère le nom du n-ième mois.

`ABMON_1 ... ABMON_12`

Récupère l'abréviation du n-ième mois.

`locale.RADIXCHAR`

Récupère le caractère de séparation *radix* (point décimal, virgule décimale, etc.).

`locale.THOUSEP`

Récupère le caractère de séparation des milliers (groupes de 3 chiffres).

`locale.YESEXPR`

Récupère une expression régulière qui peut être utilisée par la fonction *regex* pour reconnaître une réponse positive à une question fermée (oui / non).

Note : L'expression est dans une syntaxe adaptée à la fonction `regex()` de la bibliothèque C, qui peut différer de la syntaxe utilisée par `re`.

`locale.NOEXPR`

Récupère une expression régulière qui peut être utilisée par la fonction `regex(3)` pour reconnaître une réponse négative à une question fermée (oui / non).

`locale.CRNCYSTR`

Récupère le symbole monétaire, précédé de « - » si le symbole doit apparaître avant la valeur, « + » s'il doit apparaître après la valeur, ou « . » s'il doit remplacer le caractère de séparation *radix*.

`locale.ERA`

Récupère une chaîne qui représente l'ère utilisée pour le paramètre régional actuel.

La plupart des paramètres régionaux ne définissent pas cette valeur. Un exemple de région qui définit bien cette valeur est le japonais. Au Japon, la représentation traditionnelle des dates comprend le nom de l'ère correspondant au règne de l'empereur de l'époque.

Normalement, il ne devrait pas être nécessaire d'utiliser cette valeur directement. Spécifier le modificateur E dans leurs chaînes de format provoque l'utilisation de cette information par la fonction `time.strftime()`. Le format de la chaîne renvoyée n'est pas spécifié, et vous ne devez donc pas supposer en avoir connaissance sur des systèmes différents.

`locale.ERA_D_T_FMT`

Récupère la chaîne de format pour `time.strftime()` afin de représenter la date et l'heure pour un paramètre régional spécifique basée sur une ère.

`locale.ERA_D_FMT`

Récupère la chaîne de format pour `time.strftime()` afin de représenter une date pour un paramètre régional spécifique basée sur une ère.

`locale.ERA_T_FMT`

Récupère la chaîne de format pour `time.strftime()` afin de représenter une heure pour un paramètre régional spécifique basée sur une ère.

`locale.ALT_DIGITS`

Récupère une représentation de 100 valeurs maximum utilisées pour représenter les valeurs de 0 à 99.

`locale.getdefaultlocale([envvars])`

Tente de déterminer les paramètres régionaux par défaut, puis les renvoie sous la forme d'un n-uplet (code de la langue, encodage).

D'après POSIX, un programme qui n'a pas appelé `setlocale(LC_ALL, '')` fonctionne en utilisant le paramètre régional portable 'C'. Appeler `setlocale(LC_ALL, '')` lui permet d'utiliser les paramètres régionaux par défaut définis par la variable `LANG`. Comme nous ne voulons pas interférer avec les paramètres régionaux actuels, nous émuloons donc le comportement décrit ci-dessus.

Afin de maintenir la compatibilité avec d'autres plateformes, non seulement la variable `LANG` est testée, mais c'est aussi le cas pour toute une liste de variables passés en paramètre via `envvars`. La première variable à être défini sera utilisée. `envvars` utilise par défaut le chemin de recherche utilisé dans GNU gettext ; il doit toujours contenir le nom de variable 'LANG'. Le chemin de recherche de GNU gettext contient 'LC_ALL', 'LC_CTYPE', 'LANG' et 'LANGUAGE', dans cet ordre.

À l'exception du code 'C', le code d'une langue correspond à la [RFC 1766](#). Le *code de la langue* et l'*encodage* peuvent valoir `None` si leur valeur ne peut être déterminée.

`locale.getlocale(category=LC_CTYPE)`

Renvoie les réglages actuels pour la catégorie de paramètres régionaux donnée, sous la forme d'une séquence contenant le *code de la langue* et l'*encodage*. La catégorie `category` peut être l'une des valeurs `LC_*` à l'exception de `LC_ALL`. La valeur par défaut est `LC_CTYPE`.

À l'exception du code 'C', le code d'une langue correspond à la [RFC 1766](#). Le *code de la langue* et l'*encodage* peuvent valoir `None` si leur valeur ne peut être déterminée.

`locale.getpreferredencoding (do_setlocale=True)`

Renvoie le codage utilisé pour les données textuelles, selon les préférences de l'utilisateur. Les préférences de l'utilisateur sont exprimées différemment selon les systèmes et peuvent ne pas être disponibles via les interfaces de programmation sur certains systèmes. Cette fonction ne renvoie donc qu'une supposition.

Sur certains systèmes, il est nécessaire d'invoquer `setlocale()` pour obtenir les préférences de l'utilisateur, cette fonction n'est donc pas utilisable sans protection dans les programmes à fils d'exécutions multiples. Si l'appel de `setlocale` n'est pas nécessaire ou souhaité, `do_setlocale` doit être réglé à `False`.

`locale.normalize (localename)`

Renvoie un code normalisé pour le nom du paramètre régional fourni. Ce code renvoyé est structuré de façon à être utilisé avec `setlocale()`. Si la normalisation échoue, le nom d'origine est renvoyé inchangé.

Si l'encodage donné n'est pas connu, la fonction utilise l'encodage par défaut pour le code du paramètre régional, tout comme `setlocale()`.

`locale.resetlocale (category=LC_ALL)`

Définit le paramètre régional de la catégorie `category` au réglage par défaut.

Le réglage par défaut est déterminé en appelant `getdefaultlocale()`. La catégorie `category` vaut par défaut `LC_ALL`.

`locale.strcoll (string1, string2)`

Compare deux chaînes en se basant sur le paramètre `LC_COLLATE` actuel. Comme toute autre fonction de comparaison, renvoie une valeur négative, positive, ou 0, selon si `string1` est lexicographiquement inférieure, supérieure, ou égale à `string2`.

`locale.strxfrm (string)`

Transforme une chaîne de caractères en une chaîne qui peut être utilisée dans les comparaisons sensibles aux paramètres régionaux. Par exemple, `strxfrm(s1) < strxfrm(s2)` est équivalent à `strcoll(s1, s2) < 0`. Cette fonction peut être utilisée lorsque la même chaîne est comparée de façon répétitive, par exemple lors de l'assemblage d'une séquence de chaînes.

`locale.format (format, val, grouping=False, monetary=False)`

Structure un nombre `val` en fonction du paramètre `LC_NUMERIC` actuel. Le format suit les conventions de l'opérateur `%`. Pour les valeurs à virgule flottante, le point décimal est modifié si nécessaire. Si `grouping` est vrai, le regroupement est également pris en compte.

Si `monetary` est vrai, la conversion utilise un séparateur des milliers monétaire et des chaînes de regroupement.

À noter que cette fonction ne fonctionnera que pour un seul marqueur `%char`. Pour les chaînes structurées entières, utiliser `format_string()`.

`locale.format_string (format, val, grouping=False)`

Traite les marqueurs de structure en format `% val`, mais en prenant en compte les paramètres régionaux actuels.

`locale.currency (val, symbol=True, grouping=False, international=False)`

Structure un nombre `val` en fonction du paramètre `LC_MONETARY` actuel.

La chaîne renvoyée inclut le symbole monétaire si `symbol` est vrai, ce qui est le cas par défaut. Si `grouping` est vrai (ce qui n'est pas le cas par défaut), un regroupement est effectué avec la valeur. Si `international` est vrai (ce qui n'est pas le cas par défaut), le symbole de la devise internationale est utilisé.

Notez que cette fonction ne fonctionnera pas avec le paramètre régional "C", vous devez donc d'abord en définir un via `setlocale()`.

`locale.str (float)`

Structure un nombre flottant en utilisant le même format que la fonction native `str(float)`, mais en prenant en compte le point décimal.

`locale.delocalize (string)`

Convertit une chaîne de caractères en une chaîne de nombres normalisés, en suivant les réglages `LC_NUMERIC`.

Nouveau dans la version 3.5.

`locale.atoi(string)`

Convertit une chaîne de caractères en nombre à virgule flottante, en suivant les réglages `LC_NUMERIC`.

`locale.atoi(string)`

Convertit une chaîne de caractères en un entier, en suivant les réglages `LC_NUMERIC`.

`locale.LC_CTYPE`

Catégorie de paramètre régional pour les fonctions de type caractère. Suivant les réglages de la catégorie, les fonctions du module `string` gérant la casse peuvent changer leur comportement.

`locale.LC_COLLATE`

Catégorie de paramètre régional pour les tris de chaînes de caractères. Les fonctions `strcoll()` et `strxfrm()` du module `locale` sont concernées.

`locale.LC_TIME`

Catégorie de paramètre régional pour la mise en forme de la date et de l'heure. La fonction `time.strftime()` suit ces conventions.

`locale.LC_MONETARY`

Catégorie de paramètre régional pour la mise en forme des valeurs monétaires. Les options disponibles sont accessibles à partir de la fonction `localeconv()`.

`locale.LC_MESSAGES`

Catégorie de paramètre régional pour l'affichage de messages. Actuellement, Python ne gère pas les messages spécifiques aux applications qui sont sensibles aux paramètres régionaux. Les messages affichés par le système d'exploitation, comme ceux renvoyés par `os.strerror()` peuvent être affectés par cette catégorie.

`locale.LC_NUMERIC`

Catégorie de paramètre régional pour la mise en forme des nombres. Les fonctions `format()`, `atoi()`, `atof()` et `str()` du module `locale` sont affectées par cette catégorie. Toutes les autres opérations de mise en forme des nombres ne sont pas affectées.

`locale.LC_ALL`

Combinaison de tous les paramètres régionaux. Si cette option est utilisée lors du changement de paramètres régionaux, la définition de ces paramètres pour toutes les catégories est tentée. Si cela échoue pour n'importe quelle catégorie, aucune d'entre elles n'est modifiée. Lorsque les paramètres régionaux sont récupérés à l'aide de cette option, une chaîne de caractères indiquant le réglage pour toutes les catégories est renvoyée. Cette chaîne peut alors être utilisée plus tard pour restaurer les paramètres d'origine.

`locale.CHAR_MAX`

Ceci est une constante symbolique utilisée pour différentes valeurs renvoyées par `localeconv()`.

Exemple :

```
>>> import locale
>>> loc = locale.getlocale() # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xfe4n', 'foo') # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '') # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C') # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc) # restore saved locale
```

23.2.1 Contexte, détails, conseils, astuces et mises en garde

La norme C définit les paramètres régionaux comme une propriété à l'échelle d'un programme, qui peut être relativement coûteuse à changer. En plus de cela, certaines implémentations ne fonctionnent pas car des changements fréquents de paramètres régionaux peuvent causer des *core dumps*. Cela rend l'utilisation correcte de ces paramètres quelque peu pénible.

Initialement, lorsqu'un programme est démarré, les paramètres régionaux C sont utilisés, peu importe les réglages de l'utilisateur. Il y a toutefois une exception : la catégorie `LC_CTYPE` est modifiée au démarrage pour définir l'encodage des paramètres régionaux actuels comme celui défini par l'utilisateur. Le programme doit explicitement dire qu'il veut utiliser les réglages de l'utilisateur pour les autres catégories, en appelant `setlocale(LC_ALL, '')`.

C'est généralement une mauvaise idée d'appeler `setlocale()` dans une routine de bibliothèque car cela a pour effet secondaire d'affecter le programme entier. Sauvegarder et restaurer les paramètres est presque aussi mauvais : c'est coûteux et cela affecte d'autres fils d'exécutions qui s'exécutent avant que les paramètres n'aient été restaurés.

Si, lors du développement d'un module à usage général, vous avez besoin d'une version indépendante des paramètres régionaux pour une opération y étant sensible (comme c'est le cas pour certains formats utilisés avec `time.strptime()`), vous devez trouver un moyen de le faire sans utiliser la routine de la bibliothèque standard. Le mieux est encore de se convaincre que l'usage des paramètres régionaux est une bonne chose. Ce n'est qu'en dernier recours que vous devez documenter que votre module n'est pas compatible avec les réglages du paramètre régional C.

La seule façon d'effectuer des opérations numériques conformément aux paramètres régionaux est d'utiliser les fonctions spéciales définies par ce module : `atof()`, `atoi()`, `format()`, `str()`.

Il n'y a aucun moyen d'effectuer des conversions de casse et des classifications de caractères en fonction des paramètres régionaux. Pour les chaînes de caractères (Unicode), celles-ci se font uniquement en fonction de la valeur du caractère, tandis que pour les chaînes d'octets, les conversions et les classifications se font en fonction de la valeur ASCII de l'octet, et les octets dont le bit de poids fort est à 1 (c'est-à-dire les octets non ASCII) ne sont jamais convertis ou considérés comme faisant partie d'une classe de caractères comme une lettre ou une espace.

23.2.2 Pour les auteurs d'extensions et les programmes qui intègrent Python

Les modules d'extensions ne devraient jamais appeler `setlocale()`, sauf pour connaître le paramètre régional actuel. Mais comme la valeur renvoyée ne peut être utilisée que pour le restaurer, ce n'est pas très utile (sauf peut-être pour savoir si le paramètre régional est défini à C ou non).

Lorsque le code Python utilise le module `locale` pour changer le paramètre régional, cela affecte également l'application intégrée. Si l'application intégrée ne souhaite pas que cela se produise, elle doit supprimer le module d'extension `_locale` (qui fait tout le travail) de la table des modules natifs se trouvant dans le fichier `config.c`, et s'assurer que le module `_locale` n'est pas accessible en tant que bibliothèque partagée.

23.2.3 Accéder aux catalogues de messages

```
locale.gettext(msg)
locale.dgettext(domain, msg)
locale.dcgettext(domain, msg, category)
locale.textdomain(domain)
locale.bindtextdomain(domain, dir)
```

Le module `locale` expose l'interface `gettext` de la bibliothèque C sur les systèmes qui fournissent cette interface. Il se compose des fonctions `gettext()`, `dgettext()`, `dcgettext()`, `textdomain()`, `bindtextdomain()` et

`bind_textdomain_codeset()`. Elles sont similaires aux fonctions du module *gettext*, mais utilisent le format binaire de la bibliothèque C pour les catalogues de messages.

Les applications Python ne devraient normalement pas avoir besoin de faire appel à ces fonctions, mais devraient plutôt utiliser *gettext*. Une exception connue pour cette règle concerne les applications qui sont liées avec des bibliothèques C supplémentaires faisant appel à `c :func :gettext` ou `dcgettext()`. Pour ces applications, il peut être nécessaire de lier le domaine du texte, afin que les bibliothèques puissent régionaliser correctement leurs catalogues de messages.

Les modules décrits dans ce chapitre sont des *frameworks* qui encadreront la structure de vos programmes. Actuellement tous les modules décrits ici sont destinés à écrire des interfaces en ligne de commande.

La liste complète des modules décrits dans ce chapitre est :

24.1 `turtle` — Tortue graphique

Code Source : [Lib/turtle.py](#)

24.1.1 Introduction

Une façon populaire pour initier les enfants au monde du développement est le module Tortue graphique. Ce dernier faisait partie du langage de programmation Logo créé par Wally Feurzig et Seymour Papert en 1966.

Imaginez un robot sous forme de tortue partant au centre (0, 0) d'un plan cartésien x-y. Après un `import turtle`, exécutez la commande `turtle.forward(15)` et la tortue se déplace (sur l'écran) de 15 pixels en face d'elle, en dessinant une ligne.

Turtle star

La tortue permet de dessiner des formes complexes en utilisant un programme qui répète des actions élémentaires.



```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

On peut donc facilement construire des formes et images à partir de commandes simples.

Le module `turtle` est une version étendue du module homonyme appartenant à la distribution standard de Python jusqu'à la version 2.5.

Cette bibliothèque essaye de garder les avantages de l'ancien module et d'être (presque) 100% compatible avec celui-ci. Cela permet à l'apprenti développeur d'utiliser toutes les commandes, classes et méthodes de façon interactive pendant qu'il utilise le module depuis IDLE lancé avec l'option `-n`.

`Turtle` permet d'utiliser des primitives graphiques en utilisant un style de programmation orienté objet ou procédural. Du fait qu'il utilise la bibliothèque graphique `tkinter`, `Turtle` a besoin d'une version de python implémentant `Tk`.

L'interface orientée objet utilise essentiellement deux + deux classes :

1. La classe `TurtleScreen` définit une fenêtre graphique utilisé comme un terrain de jeu pour les dessins de la tortue. Le constructeur de cette classe a besoin d'un `tkinter.Canvas` ou `ScrolledCanvas` comme argument. Cette classe doit être utilisée seulement si `turtle` fait partie intégrante d'une autre application. La fonction `Screen()` renvoie un singleton d'une sous-classe de `TurtleScreen`. Elle doit être utilisée quand le module `turtle` est utilisé de façon autonome pour dessiner. Le singleton renvoyé ne peut hériter de sa classe. Toutes les méthodes de `TurtleScreen/Screen` existent également sous la forme de fonctions, c'est-à-dire que ces dernières peuvent être utilisées dans un style procédural.
2. La classe `RawTurtle` (alias `RawPen`) définit des objets `Turtle` qui peuvent dessiner sur la classe `TurtleScreen`. Son constructeur prend en paramètre un `Canvas`, un `ScrolledCanvas` ou un `TurtleScreen` permettant à l'objet `RawTurtle` de savoir où écrire. La sous-classe `Turtle` (alias : `Pen`), dérivée de `RawTurtle`, dessine sur l'instance `Screen` qui est créée automatiquement si elle n'est pas déjà présente.

Toutes les méthodes de *RawTurtle/Turtle* existent également sous la forme de fonctions, c'est-à-dire que ces dernières pourront être utilisées en style procédural.

L'interface procédurale met à disposition des fonctions équivalentes à celles des méthodes des classes *Screen* et *Turtle*. Le nom d'une fonction est le même que la méthode équivalente. Un objet *Screen* est créé automatiquement dès qu'une fonction dérivée d'une méthode *Screen* est appelée. Un objet *Turtle* (sans nom) est créé automatiquement dès qu'une fonction dérivée d'une méthode *Turtle* est appelée.

Afin de pouvoir utiliser plusieurs tortues simultanément sur l'écran, vous devez utiliser l'interface orientée-objet.

Note : La liste des paramètres des fonctions est donnée dans cette documentation. Les méthodes ont, évidemment, le paramètre *self* comme premier argument, mais ce dernier n'est pas indiqué ici.

24.1.2 Résumé des méthodes de *Turtle* et *Screen*

Les méthodes du module *Turtle*

Les mouvements dans le module *Turtle*

Bouger et dessiner

```
forward() | fd()
backward() | bk() | back()
right() | rt()
left() | lt()
goto() | setpos() | setposition()
setx()
sety()
setheading() | seth()
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()
```

Tell Turtle's state

```
position() | pos()
towards()
xcor()
ycor()
heading()
distance()
```

Setting and measurement

```
degrees()
radians()
```

Pen control

Drawing state

```
pendown() | pd() | down()
```

```
penup() | pu() | up()
pensize() | width()
pen()
isdown()
```

Color control

```
color()
pencolor()
fillcolor()
```

Filling

```
filling()
begin_fill()
end_fill()
```

More drawing control

```
reset()
clear()
write()
```

Turtle state

Visibility

```
showturtle() | st()
hideturtle() | ht()
isvisible()
```

Appearance

```
shape()
resizemode()
shapeseize() | turtlesize()
shearfactor()
settiltangle()
tiltangle()
tilt()
shapetransform()
get_shapepoly()
```

Using events

```
onclick()
onrelease()
ondrag()
```

Special Turtle methods

```
begin_poly()
end_poly()
get_poly()
clone()
getturtle() | getpen()
getscreen()
setundobuffer()
undobufferentries()
```

Methods of TurtleScreen/Screen

Window control

```
bgcolor()  
bgpic()  
clear() | clearscreen()  
reset() | resetscreen()  
screensize()  
setworldcoordinates()
```

Animation control

```
delay()  
tracer()  
update()
```

Using screen events

```
listen()  
onkey() | onkeyrelease()  
onkeypress()  
onclick() | onscreenclick()  
ontimer()  
mainloop() | done()
```

Settings and special methods

```
mode()  
colormode()  
getcanvas()  
getshapes()  
register_shape() | addshape()  
turtles()  
window_height()  
window_width()
```

Input methods

```
textinput()  
numinput()
```

Methods specific to Screen

```
bye()  
exitonclick()  
setup()  
title()
```

24.1.3 Methods of RawTurtle/Turtle and corresponding functions

Most of the examples in this section refer to a Turtle instance called `turtle`.

Les mouvements dans le module *Turtle*

`turtle.forward(distance)`

`turtle.fd(distance)`

Paramètres *distance* – a number (integer or float)

Move the turtle forward by the specified *distance*, in the direction the turtle is headed.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.back(distance)`

`turtle.bk(distance)`

`turtle.backward(distance)`

Paramètres *distance* – a number

Move the turtle backward by *distance*, opposite to the direction the turtle is headed. Do not change the turtle's heading.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.right(angle)`

`turtle.rt(angle)`

Paramètres *angle* – a number (integer or float)

Turn turtle right by *angle* units. (Units are by default degrees, but can be set via the `degrees()` and `radians()` functions.) Angle orientation depends on the turtle mode, see `mode()`.

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

`turtle.left(angle)`

`turtle.lt(angle)`

Paramètres *angle* – a number (integer or float)

Turn turtle left by *angle* units. (Units are by default degrees, but can be set via the `degrees()` and `radians()` functions.) Angle orientation depends on the turtle mode, see `mode()`.

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

`turtle.goto(x, y=None)`

`turtle.setpos(x, y=None)`

`turtle.setposition(x, y=None)`

Paramètres

- **x** – a number or a pair/vector of numbers
- **y** – a number or None

If *y* is None, *x* must be a pair of coordinates or a *Vec2D* (e.g. as returned by `pos()`).

Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle's orientation.

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

`turtle.setx(x)`

Paramètres x – a number (integer or float)

Set the turtle's first coordinate to *x*, leave second coordinate unchanged.

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

`turtle.sety(y)`

Paramètres y – a number (integer or float)

Set the turtle's second coordinate to *y*, leave first coordinate unchanged.

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

`turtle.setheading(to_angle)`

`turtle.seth(to_angle)`

Paramètres to_angle – a number (integer or float)

Set the orientation of the turtle to *to_angle*. Here are some common directions in degrees :

standard mode	logo mode
0 - east	0 - north
90 - north	90 - east
180 - west	180 - south
270 - south	270 - west

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

`turtle.home()`

Move turtle to the origin – coordinates (0,0) – and set its heading to its start-orientation (which depends on the mode, see `mode()`).

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

`turtle.circle(radius, extent=None, steps=None)`

Paramètres

- **radius** – a number
- **extent** – a number (or None)
- **steps** – an integer (or None)

Draw a circle with given *radius*. The center is *radius* units left of the turtle; *extent* – an angle – determines which part of the circle is drawn. If *extent* is not given, draw the entire circle. If *extent* is not a full circle, one endpoint of the arc is the current pen position. Draw the arc in counterclockwise direction if *radius* is positive, otherwise in clockwise direction. Finally the direction of the turtle is changed by the amount of *extent*.

As the circle is approximated by an inscribed regular polygon, *steps* determines the number of steps to use. If not given, it will be calculated automatically. May be used to draw regular polygons.

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180) # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

`turtle.dot(size=None, *color)`

Paramètres

- **size** – an integer ≥ 1 (if given)
- **color** – a colorstring or a numeric color tuple

Draw a circular dot with diameter *size*, using *color*. If *size* is not given, the maximum of `pensize+4` and `2*pensize` is used.

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

`turtle.stamp()`

Stamp a copy of the turtle shape onto the canvas at the current turtle position. Return a `stamp_id` for that stamp, which can be used to delete it by calling `clearstamp(stamp_id)`.

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

`turtle.clearstamp(stampid)`

Paramètres `stampid` – an integer, must be return value of previous `stamp()` call

Delete stamp with given `stampid`.

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

`turtle.clearstamps(n=None)`

Paramètres `n` – an integer (or None)

Delete all or first/last `n` of turtle's stamps. If `n` is None, delete all stamps, if `n > 0` delete first `n` stamps, else if `n < 0` delete last `n` stamps.

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
19
20
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

`turtle.undo()`

Undo (repeatedly) the last turtle action(s). Number of available undo actions is determined by the size of the `undobuffer`.

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

`turtle.speed(speed=None)`

Paramètres `speed` – an integer in the range 0..10 or a speedstring (see below)

Set the turtle's speed to an integer value in the range 0..10. If no argument is given, return current speed.

If input is a number greater than 10 or smaller than 0.5, speed is set to 0. Speedstrings are mapped to speedvalues as follows :

- « fastest » : 0
- « fast » : 10
- « normal » : 6
- « slow » : 3
- « slowest » : 1

Speeds from 1 to 10 enforce increasingly faster animation of line drawing and turtle turning.

Attention : *speed* = 0 means that *no* animation takes place. forward/back makes turtle jump and likewise left/right make the turtle turn instantly.

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

Tell Turtle's state

`turtle.position()`

`turtle.pos()`

Return the turtle's current location (x,y) (as a *Vec2D* vector).

```
>>> turtle.pos()
(440.00, -0.00)
```

`turtle.towards(x, y=None)`

Paramètres

- **x** – a number or a pair/vector of numbers or a turtle instance
- **y** – a number if *x* is a number, else None

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle's start orientation which depends on the mode - « standard »/ « world » or « logo »).

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0, 0)
225.0
```

`turtle.xcor()`

Return the turtle's x coordinate.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28, 76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

`turtle.ycor()`

Return the turtle's y coordinate.

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00, 86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.heading()

Return the turtle's current heading (value depends on the turtle mode, see [mode\(\)](#)).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.distance(x, y=None)

Paramètres

- **x** – a number or a pair/vector of numbers or a turtle instance
- **y** – a number if *x* is a number, else None

Return the distance from the turtle to (x,y), the given vector, or the given other turtle, in turtle step units.

```
>>> turtle.home()
>>> turtle.distance(30, 40)
50.0
>>> turtle.distance((30, 40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

Settings for measurement

turtle.degrees(fullcircle=360.0)

Paramètres **fullcircle** – a number

Set angle measurement units, i.e. set number of « degrees » for a full circle. Default value is 360 degrees.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.radians()

Set the angle measurement units to radians. Equivalent to `degrees(2*math.pi)`.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Pen control

Drawing state

```
turtle.pendown()
turtle.pd()
turtle.down()
```

Pull the pen down – drawing when moving.

```
turtle.penup()
turtle.pu()
turtle.up()
```

Pull the pen up – no drawing when moving.

```
turtle.pensize(width=None)
turtle.width(width=None)
```

Paramètres **width** – a positive number

Set the line thickness to *width* or return it. If *resizemode* is set to « auto » and *turtleshape* is a polygon, that polygon is drawn with the same line thickness. If no argument is given, the current pensize is returned.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)    # from here on lines of width 10 are drawn
```

```
turtle.pen(pen=None, **pendict)
```

Paramètres

- **pen** – a dictionary with some or all of the below listed keys
- **pendict** – one or more keyword-arguments with the below listed keys as keywords

Return or set the pen's attributes in a « pen-dictionary » with the following key/value pairs :

- « shown » : True/False
- « pendown » : True/False
- « pencolor » : color-string or color-tuple
- « fillcolor » : color-string or color-tuple
- « pensize » : positive number
- « speed » : number in range 0..10
- « resizingmode » : « auto » or « user » or « noresizing »
- « stretchfactor » : (positive number, positive number)
- « outline » : positive number
- « tilt » : number

This dictionary can be used as argument for a subsequent call to `pen()` to restore the former pen-state. Moreover one or more of these attributes can be provided as keyword-arguments. This can be used to set several pen attributes in one statement.

```

>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]

```

`turtle.isdown()`

Return True if pen is down, False if it's up.

```

>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True

```

Color control

`turtle.pencolor(*args)`

Return or set the pencolor.

Four input formats are allowed :

pencolor() Return the current pencolor as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor call.

pencolor(colorstring) Set pencolor to *colorstring*, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

pencolor(r, g, b) Set pencolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see `colormode()`).

pencolor(r, g, b)

Set pencolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If `turtleshape` is a polygon, the outline of that polygon is drawn with the newly set pencolor.

```

>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)

```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.fillcolor(*args)

Return or set the fillcolor.

Four input formats are allowed :

fillcolor() Return the current fillcolor as color specification string, possibly in tuple format (see example).

May be used as input to another color/pencolor/fillcolor call.

fillcolor(colorstring) Set fillcolor to *colorstring*, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

fillcolor(r, g, b) Set fillcolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see *colormode()*).

fillcolor(r, g, b)

Set fillcolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the interior of that polygon is drawn with the newly set fillcolor.

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> col = turtle.pencolor()
>>> col
(50.0, 193.0, 143.0)
>>> turtle.fillcolor(col)
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.color(*args)

Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows :

color() Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by *pencolor()* and *fillcolor()*.

color(colorstring), color(r, g, b), color(r, g, b) Inputs as in *pencolor()*, set both, fillcolor and pencolor, to the given value.

color(colorstring1, colorstring2), color(r1, g1, b1), (r2, g2, b2)

Equivalent to *pencolor(colorstring1)* and *fillcolor(colorstring2)* and analogously if the other input format is used.

If turtleshape is a polygon, outline and interior of that polygon is drawn with the newly set colors.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

See also : Screen method *colormode()*.

Filling

`turtle.filling()`

Return fillstate (True if filling, False else).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

`turtle.begin_fill()`

To be called just before drawing a shape to be filled.

`turtle.end_fill()`

Fill the shape drawn after the last call to `begin_fill()`.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

More drawing control

`turtle.reset()`

Delete the turtle's drawings from the screen, re-center the turtle and set variables to the default values.

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

`turtle.clear()`

Delete the turtle's drawings from the screen. Do not move turtle. State and position of the turtle as well as drawings of other turtles are not affected.

`turtle.write(arg, move=False, align="left", font=("Arial", 8, "normal"))`

Paramètres

- **arg** – object to be written to the TurtleScreen
- **move** – True/False
- **align** – one of the strings « left », « center » or right »
- **font** – a triple (fontname, fontsize, fonttype)

Write text - the string representation of *arg* - at the current turtle position according to *align* (« left », « center » or right) and with the given font. If *move* is true, the pen is moved to the bottom-right corner of the text. By default, *move* is False.

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

Turtle state

Visibility

`turtle.hideturtle()`

`turtle.ht()`

Make the turtle invisible. It's a good idea to do this while you're in the middle of doing some complex drawing, because hiding the turtle speeds up the drawing observably.

```
>>> turtle.hideturtle()
```

`turtle.showturtle()`

`turtle.st()`

Make the turtle visible.

```
>>> turtle.showturtle()
```

`turtle.isvisible()`

Return True if the Turtle is shown, False if it's hidden.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

Appearance

`turtle.shape(name=None)`

Paramètres `name` – a string which is a valid shapename

Set turtle shape to shape with given *name* or, if *name* is not given, return name of current shape. Shape with *name* must exist in the TurtleScreen's shape dictionary. Initially there are the following polygon shapes : « arrow », « turtle », « circle », « square », « triangle », « classic ». To learn about how to deal with shapes see Screen method `register_shape()`.

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

`turtle.resizemode(rmode=None)`

Paramètres `rmode` – one of the strings « auto », « user », « noresize »

Set `resizemode` to one of the values : « auto », « user », « noresize ». If *rmode* is not given, return current `resizemode`. Different `resizemodes` have the following effects :

- « auto » : adapts the appearance of the turtle corresponding to the value of `pensize`.
- « user » : adapts the appearance of the turtle according to the values of `stretchfactor` and `outlinewidth` (outline), which are set by `shapeseize()`.
- « noresize » : no adaption of the turtle's appearance takes place.

`resizemode`(« user ») is called by `shapeseize()` when used with arguments.

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

`turtle.shapesize(stretch_wid=None, stretch_len=None, outline=None)`
`turtle.turtlesize(stretch_wid=None, stretch_len=None, outline=None)`

Paramètres

- **stretch_wid** – positive number
- **stretch_len** – positive number
- **outline** – positive number

Return or set the pen's attributes x/y-stretchfactors and/or outline. Set `resizemode` to « user ». If and only if `resizemode` is set to « user », the turtle will be displayed stretched according to its stretchfactors : *stretch_wid* is stretchfactor perpendicular to its orientation, *stretch_len* is stretchfactor in direction of its orientation, *outline* determines the width of the shapes's outline.

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

`turtle.shearfactor(shear=None)`

Paramètres **shear** – number (optional)

Set or return the current shearfactor. Shear the turtleshape according to the given shearfactor *shear*, which is the tangent of the shear angle. Do *not* change the turtle's heading (direction of movement). If *shear* is not given : return the current shearfactor, i. e. the tangent of the shear angle, by which lines parallel to the heading of the turtle are sheared.

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

`turtle.tilt(angle)`

Paramètres **angle** – a number

Rotate the turtleshape by *angle* from its current tilt-angle, but do *not* change the turtle's heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

`turtle.settiltangle(angle)`

Paramètres **angle** – a number

Rotate the turtleshape to point in the direction specified by *angle*, regardless of its current tilt-angle. *Do not* change the turtle's heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

Obsolète depuis la version 3.1.

`turtle.tiltangle` (*angle=None*)

Paramètres *angle* – a number (optional)

Set or return the current tilt-angle. If *angle* is given, rotate the turtleshape to point in the direction specified by *angle*, regardless of its current tilt-angle. *Do not* change the turtle's heading (direction of movement). If *angle* is not given : return the current tilt-angle, i. e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

`turtle.shapetransform` (*t11=None, t12=None, t21=None, t22=None*)

Paramètres

- *t11* – a number (optional)
- *t12* – a number (optional)
- *t21* – a number (optional)
- *t22* – a number (optional)

Set or return the current transformation matrix of the turtle shape.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtleshape according to the matrix consisting of first row *t11*, *t12* and second row *t21*, *t22*. The determinant $t11 * t22 - t12 * t21$ must not be zero, otherwise an error is raised. Modify stretchfactor, shearfactor and tiltangle according to the given matrix.

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

`turtle.get_shapepoly` ()

Return the current shape polygon as tuple of coordinate pairs. This can be used to define a new shape or components of a compound shape.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

Using events

`turtle.onclick(fun, btn=1, add=None)`

Paramètres

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this turtle. If *fun* is None, existing bindings are removed. Example for the anonymous turtle, i.e. the procedural way :

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

`turtle.onrelease(fun, btn=1, add=None)`

Paramètres

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-button-release events on this turtle. If *fun* is None, existing bindings are removed.

```
>>> class MyTurtle(Turtle):
...     def glow(self, x, y):
...         self.fillcolor("red")
...     def unglow(self, x, y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)  # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

`turtle.ondrag(fun, btn=1, add=None)`

Paramètres

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-move events on this turtle. If *fun* is None, existing bindings are removed.

Remark : Every sequence of mouse-move-events on a turtle is preceded by a mouse-click event on that turtle.

```
>>> turtle.ondrag(turtle.goto)
```

Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

Special Turtle methods

`turtle.begin_poly()`

Start recording the vertices of a polygon. Current turtle position is first vertex of polygon.

`turtle.end_poly()`

Stop recording the vertices of a polygon. Current turtle position is last vertex of polygon. This will be connected with the first vertex.

`turtle.get_poly()`

Return the last recorded polygon.

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

`turtle.clone()`

Create and return a clone of the turtle with same position, heading and turtle properties.

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

`turtle.getturtle()`

`turtle.getpen()`

Return the Turtle object itself. Only reasonable use : as a function to return the « anonymous turtle » :

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

`turtle.getscreen()`

Return the *TurtleScreen* object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

`turtle.setundobuffer(size)`

Paramètres *size* – an integer or None

Set or disable undobuffer. If *size* is an integer an empty undobuffer of given size is installed. *size* gives the maximum number of turtle actions that can be undone by the *undo()* method/function. If *size* is None, the undobuffer is disabled.

```
>>> turtle.setundobuffer(42)
```

`turtle.undobufferentries()`

Return number of entries in the undobuffer.

```
>>> while undobufferentries():
...     undo()
```

Compound shapes

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class *Shape* explicitly as described below :

1. Create an empty Shape object of type « compound ».
2. Add as many components to this object as desired, using the `addcomponent()` method.

Par exemple :

```
>>> s = Shape("compound")
>>> poly1 = ((0,0), (10,-5), (0,10), (-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0), (10,-5), (-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. Now add the Shape to the Screen's shapelist and use it :

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

Note : The *Shape* class is used internally by the `register_shape()` method in different ways. The application programmer has to deal with the Shape class *only* when using compound shapes like shown above !

24.1.4 Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called `screen`.

Window control

`turtle.bgcolor(*args)`

Paramètres `args` – a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers

Set or return background color of the TurtleScreen.

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

`turtle.bgpic(picname=None)`

Paramètres `picname` – a string, name of a gif-file or "nopic", or None

Set background image or return name of current backgroundimage. If `picname` is a filename, set the corresponding image as background. If `picname` is "nopic", delete background image, if present. If `picname` is None, return the filename of the current backgroundimage.

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.clear()`

`turtle.clearscreen()`

Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state : white background, no background image, no event bindings and tracing on.

Note : This TurtleScreen method is available as a global function only under the name `clearscreen`. The global function `clear` is a different one derived from the Turtle method `clear`.

`turtle.reset()`

`turtle.resetscreen()`

Reset all Turtles on the Screen to their initial state.

Note : This TurtleScreen method is available as a global function only under the name `resetscreen`. The global function `reset` is another one derived from the Turtle method `reset`.

`turtle.screensize (canvwidth=None, canvheight=None, bg=None)`

Paramètres

- **canvwidth** – positive integer, new width of canvas in pixels
- **canvheight** – positive integer, new height of canvas in pixels
- **bg** – colorstring or color-tuple, new background color

If no arguments are given, return current (canvaswidth, canvasheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

e.g. to search for an erroneously escaped turtle;-)

`turtle.setworldcoordinates (llx, lly, urx, ury)`

Paramètres

- **llx** – a number, x-coordinate of lower left corner of canvas
- **lly** – a number, y-coordinate of lower left corner of canvas
- **urx** – a number, x-coordinate of upper right corner of canvas
- **ury** – a number, y-coordinate of upper right corner of canvas

Set up user-defined coordinate system and switch to mode « world » if necessary. This performs a `screen.reset()`. If mode « world » is already active, all drawings are redrawn according to the new coordinates.

ATTENTION : in user-defined coordinate systems angles may appear distorted.

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
... 
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> for _ in range(8):
...     left(45); fd(2)    # a regular octagon
```

Animation control

`turtle.delay` (*delay=None*)

Paramètres `delay` – positive integer

Set or return the drawing *delay* in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Optional argument :

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

`turtle.tracer` (*n=None, delay=None*)

Paramètres

- `n` – nonnegative integer
- `delay` – nonnegative integer

Turn turtle animation on/off and set delay for update drawings. If *n* is given, only each *n*-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) When called without arguments, returns the currently stored value of *n*. Second argument sets delay value (see `delay()`).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

`turtle.update` ()

Perform a TurtleScreen update. To be used when tracer is turned off.

See also the RawTurtle/Turtle method `speed()`.

Using screen events

`turtle.listen` (*xdummy=None, ydummy=None*)

Set focus on TurtleScreen (in order to collect key-events). Dummy arguments are provided in order to be able to pass `listen()` to the onclick method.

`turtle.onkey` (*fun, key*)

`turtle.onkeyrelease` (*fun, key*)

Paramètres

- `fun` – a function with no arguments or None
- `key` – a string : key (e.g. « a ») or key-symbol (e.g. « space »)

Bind *fun* to key-release event of *key*. If *fun* is None, event bindings are removed. Remark : in order to be able to register key-events, TurtleScreen must have the focus. (See method `listen()`.)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

`turtle.onkeypress` (*fun*, *key=None*)

Paramètres

- **fun** – a function with no arguments or `None`
- **key** – a string : key (e.g. « a ») or key-symbol (e.g. « space »)

Bind *fun* to key-press event of key if key is given, or to any key-press-event if no key is given. Remark : in order to be able to register key-events, TurtleScreen must have focus. (See method `listen()`.)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

`turtle.onclick` (*fun*, *btn=1*, *add=None*)

`turtle.onscreenclick` (*fun*, *btn=1*, *add=None*)

Paramètres

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this screen. If *fun* is `None`, existing bindings are removed.

Example for a TurtleScreen instance named `screen` and a Turtle instance named `turtle` :

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen will
>>>                               # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

Note : This TurtleScreen method is available as a global function only under the name `onscreenclick`. The global function `onclick` is another one derived from the Turtle method `onclick`.

`turtle.ontimer` (*fun*, *t=0*)

Paramètres

- **fun** – a function with no arguments
- **t** – a number ≥ 0

Install a timer that calls *fun* after *t* milliseconds.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()    ### makes the turtle march around
>>> running = False
```

```
turtle.mainloop()
turtle.done()
```

Starts event loop - calling Tkinter's mainloop function. Must be the last statement in a turtle graphics program. Must *not* be used if a script is run from within IDLE in -n mode (No subprocess) - for interactive use of turtle graphics.

```
>>> screen.mainloop()
```

Input methods

```
turtle.textinput (title, prompt)
```

Paramètres

- **title** – string
- **prompt** – string

Pop up a dialog window for input of a string. Parameter title is the title of the dialog window, prompt is a text mostly describing what information to input. Return the string input. If the dialog is canceled, return None.

```
>>> screen.textinput("NIM", "Name of first player:")
```

```
turtle.numinput (title, prompt, default=None, minval=None, maxval=None)
```

Paramètres

- **title** – string
- **prompt** – string
- **default** – number (optional)
- **minval** – number (optional)
- **maxval** – number (optional)

Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default : default value, minval : minimum value for input, maxval : maximum value for input The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the number input. If the dialog is canceled, return None.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

Settings and special methods

```
turtle.mode (mode=None)
```

Paramètres mode – one of the strings « standard », « logo » or « world »

Set turtle mode (« standard », « logo » or « world ») and perform reset. If mode is not given, current mode is returned.

Mode « standard » is compatible with old *turtle*. Mode « logo » is compatible with most Logo turtle graphics. Mode « world » uses user-defined « world coordinates ». **Attention** : in this mode angles appear distorted if x/y unit-ratio doesn't equal 1.

Mode	Initial turtle heading	positive angles
« standard »	to the right (east)	counterclockwise
« logo »	upward (north)	clockwise

```
>>> mode("logo")    # resets turtle heading to north
>>> mode()
'logo'
```

`turtle.colormode (cmode=None)`

Paramètres `cmode` – one of the values 1.0 or 255

Return the colormode or set it to 1.0 or 255. Subsequently *r*, *g*, *b* values of color triples have to be in the range 0..*cmode*.

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

`turtle.getcanvas ()`

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

`turtle.getshapes ()`

Return a list of names of all currently available turtle shapes.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

`turtle.register_shape (name, shape=None)`

`turtle.addshape (name, shape=None)`

There are three different ways to call this function :

(1) *name* is the name of a gif-file and *shape* is None : Install the corresponding image shape.

```
>>> screen.register_shape("turtle.gif")
```

Note : Image shapes *do not* rotate when turning the turtle, so they do not display the heading of the turtle !

(2) *name* is an arbitrary string and *shape* is a tuple of pairs of coordinates : Install the corresponding polygon shape.

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(3) *name* is an arbitrary string and *shape* is a (compound) *Shape* object : Install the corresponding compound shape.

Add a turtle shape to TurtleScreen's shapelist. Only thusly registered shapes can be used by issuing the command `shape (shapename)`.

`turtle.turtles ()`

Return the list of turtles on the screen.

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.window_height ()`

Return the height of the turtle window.

```
>>> screen.window_height()
480
```

`turtle.window_width()`
Return the width of the turtle window.

```
>>> screen.window_width()
640
```

Methods specific to Screen, not inherited from TurtleScreen

`turtle.bye()`
Shut the turtlegraphics window.

`turtle.exitonclick()`
Bind `bye()` method to mouse clicks on the Screen.
If the value « `using_IDLE` » in the configuration dictionary is `False` (default value), also enter `mainloop`. Remark : If IDLE with the `-n` switch (no subprocess) is used, this value should be set to `True` in `turtle.cfg`. In this case IDLE's own `mainloop` is active also for the client script.

`turtle.setup(width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"])`
Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a `turtle.cfg` file.

Paramètres

- **width** – if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen
- **height** – if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen
- **startx** – if positive, starting position in pixels from the left edge of the screen, if negative from the right edge, if `None`, center window horizontally
- **starty** – if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if `None`, center window vertically

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>                 # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>                 # sets window to 75% of screen by 50% of screen and centers
```

`turtle.title(titlestring)`

Paramètres `titlestring` – a string that is shown in the titlebar of the turtle graphics window
Set title of turtle window to `titlestring`.

```
>>> screen.title("Welcome to the turtle zoo!")
```

24.1.5 Public classes

class `turtle.RawTurtle` (*canvas*)

class `turtle.RawPen` (*canvas*)

Paramètres **canvas** – a `tkinter.Canvas`, a *ScrolledCanvas* or a *TurtleScreen*

Create a turtle. The turtle has all methods described above as « methods of Turtle/RawTurtle ».

class `turtle.Turtle`

Subclass of `RawTurtle`, has the same interface but draws on a default *Screen* object created automatically when needed for the first time.

class `turtle.TurtleScreen` (*cv*)

Paramètres **cv** – a `tkinter.Canvas`

Provides screen oriented methods like `setbg()` etc. that are described above.

class `turtle.Screen`

Subclass of `TurtleScreen`, with *four methods added*.

class `turtle.ScrolledCanvas` (*master*)

Paramètres **master** – some Tkinter widget to contain the `ScrolledCanvas`, i.e. a Tkinter-canvas with scrollbars added

Used by class `Screen`, which thus automatically provides a `ScrolledCanvas` as playground for the turtles.

class `turtle.Shape` (*type_, data*)

Paramètres **type_** – one of the strings « polygon », « image », « compound »

Data structure modeling shapes. The pair (*type_*, *data*) must follow this specification :

<i>type_</i>	<i>data</i>
« polygon »	a polygon-tuple, i.e. a tuple of pairs of coordinates
« image »	an image (in this form only used internally !)
« compound »	None (a compound shape has to be constructed using the <i>addcomponent()</i> method)

addcomponent (*poly*, *fill*, *outline=None*)

Paramètres

- **poly** – a polygon, i.e. a tuple of pairs of numbers
- **fill** – a color the *poly* will be filled with
- **outline** – a color for the poly's outline (if given)

Exemple :

```
>>> poly = ((0,0), (10,-5), (0,10), (-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

See *Compound shapes*.

class `turtle.Vec2D` (*x*, *y*)

A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics programs too. Derived from tuple, so a vector is a tuple !

Provides (for *a*, *b* vectors, *k* number) :

- *a* + *b* vector addition
- *a* - *b* vector subtraction
- *a* * *b* inner product
- *k* * *a* and *a* * *k* multiplication with scalar
- `abs(a)` absolute value of *a*
- `a.rotate(angle)` rotation

24.1.6 Help and configuration

How to use help

The public methods of the `Screen` and `Turtle` classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities :

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.
- Calling `help()` on methods or functions displays the docstrings :

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    >>> screen.bgcolor("orange")
    >>> screen.bgcolor()
    "orange"
    >>> screen.bgcolor(0.5,0,0.5)
    >>> screen.bgcolor()
    "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- The docstrings of the functions which are derived from methods have a modified form :

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

    >>> bgcolor("orange")
    >>> bgcolor()
    "orange"
    >>> bgcolor(0.5,0,0.5)
    >>> bgcolor()
    "#800080"
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

These modified docstrings are created automatically together with the function definitions that are derived from the methods at import time.

Translation of docstrings into different languages

There is a utility to create a dictionary the keys of which are the method names and the values of which are the docstrings of the public methods of the classes `Screen` and `Turtle`.

`turtle.write_docstringdict(filename="turtle_docstringdict")`

Paramètres `filename` – a string, used as filename

Create and write docstring-dictionary to a Python script with the given filename. This function has to be called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to the Python script `filename.py`. It is intended to serve as a template for translation of the docstrings into different languages.

If you (or your students) want to use `turtle` with online help in your native language, you have to translate the docstrings and save the resulting file as e.g. `turtle_docstringdict_german.py`.

If you have an appropriate entry in your `turtle.cfg` file this dictionary will be read in at import time and will replace the original English docstrings.

At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to glingsl@aon.at.)

How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to your needs, e.g. for use in a classroom, you can prepare a configuration file `turtle.cfg` which will be read at import time and modify the configuration according to its settings.

The built in configuration would correspond to the following `turtle.cfg` :

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
```

(suite sur la page suivante)

(suite de la page précédente)

```

undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False

```

Short explanation of selected entries :

- The first four lines correspond to the arguments of the `Screen.setup()` method.
- Line 5 and 6 correspond to the arguments of the method `Screen.screensize()`.
- *shape* can be any of the built-in shapes, e.g : arrow, turtle, etc. For more info try `help(shape)`.
- If you want to use no fillcolor (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the `cfg`-file).
- If you want to reflect the turtle its state, you have to use `resizemode = auto`.
- If you set e.g. `language = italian` the `docstringdict turtle_docstringdict_italian.py` will be loaded at import time (if present on the import path, e.g. in the same directory as `turtle`).
- The entries `exampleturtle` and `examplescreen` define the names of these objects as they occur in the docstrings. The transformation of method-docstrings to function-docstrings will delete these names from the docstrings.
- `using_IDLE` : Set this to `True` if you regularly work with IDLE and its `-n` switch (« no subprocess »). This will prevent `exitonclick()` to enter the mainloop.

There can be a `turtle.cfg` file in the directory where `turtle` is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The `Lib/turtledemo` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).

24.1.7 turtledemo — Demo scripts

The `turtledemo` package includes a set of demo scripts. These scripts can be run and viewed using the supplied demo viewer as follows :

```
python -m turtledemo
```

Alternatively, you can run the demo scripts individually. For example,

```
python -m turtledemo.bytedesign
```

The `turtledemo` package directory contains :

- A demo viewer `__main__.py` which can be used to view the sourcecode of the scripts and run them at the same time.
- Multiple scripts demonstrating different features of the `turtle` module. Examples can be accessed via the Examples menu. They can also be run standalone.
- A `turtle.cfg` file which serves as an example of how to write and use such files.

The demo scripts are :

Nom	Description	Features
bytedesign	complex classical turtle graphics pattern	<code>tracer()</code> , <code>delay</code> , <code>update()</code>
chaos	graphs Verhulst dynamics, shows that computer's computations can generate results sometimes against the common sense expectations	world coordinates
clock	analog clock showing time of your computer	turtles as clock's hands, <code>ontimer</code>
colormixer	experiment with r, g, b	<code>ondrag()</code>
forest	3 breadth-first trees	randomization
fractalcurves	Hilbert & Koch curves	recursion
lindenmayer	ethnomathematics (indian kolams)	L-System
minimal_hanoi	Towers of Hanoi	Rectangular Turtles as Hanoi discs (shape, shapsize)
nim	play the classical nim game with three heaps of sticks against the computer.	turtles as nimsticks, event driven (mouse, keyboard)
paint	super minimalistic drawing program	<code>onclick()</code>
peace	elementary	turtle : appearance and animation
penrose	aperiodic tiling with kites and darts	<code>stamp()</code>
planet_and_moon	simulation of gravitational system	compound shapes, <code>Vec2D</code>
round_dance	dancing turtles rotating pairwise in opposite direction	compound shapes, clone shapsize, tilt, <code>get_shapepoly</code> , <code>update</code>
sorting_animate	visual demonstration of different sorting methods	simple alignment, randomization
tree	a (graphical) breadth first tree (using generators)	<code>clone()</code>
two_canvases	simple design	turtles on two canvases
wikipedia	a pattern from the wikipedia article on turtle graphics	<code>clone()</code> , <code>undo()</code>
yinyang	another elementary example	<code>circle()</code>

Have fun !

24.1.8 Changes since Python 2.6

- The methods `Turtle.tracer()`, `Turtle.window_width()` and `Turtle.window_height()` have been eliminated. Methods with these names and functionality are now available only as methods of `Screen`. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding `TurtleScreen/Screen`-methods.)
- The method `Turtle.fill()` has been eliminated. The behaviour of `begin_fill()` and `end_fill()` have changed slightly : now every filling-process must be completed with an `end_fill()` call.
- A method `Turtle.filling()` has been added. It returns a boolean value : `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

24.1.9 Changes since Python 3.0

- The methods `Turtle.shearfactor()`, `Turtle.shapetransform()` and `Turtle.get_shapepoly()` have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. `Turtle.tiltangle()` has been enhanced in functionality : it now can be used to get or set the tiltangle. `Turtle.settiltangle()` has been deprecated.
- The method `Screen.onkeypress()` has been added as a complement to `Screen.onkey()` which in fact binds actions to the keyrelease event. Accordingly the latter has got an alias : `Screen.onkeyrelease()`.
- The method `Screen.mainloop()` has been added. So when working only with `Screen` and `Turtle` objects one must not additionally import `mainloop()` anymore.
- Two input methods has been added `Screen.textinput()` and `Screen.numinput()`. These popup input dialogs and return strings and numbers respectively.
- Two example scripts `tdemo_nim.py` and `tdemo_round_dance.py` have been added to the `Lib/turtledemo` directory.

24.2 cmd — Interpréteurs en ligne de commande.

Code source : [Lib/cmd.py](#)

La `Cmd` fournit une base simple permettant d'écrire des interpréteurs en ligne de commande. Ceux-ci sont souvent utiles pour piloter des tests, pour des outils administratifs, et pour des prototypes destinés à être intégrés à une interface plus sophistiquée.

class `cmd.Cmd` (*completekey='tab', stdin=None, stdout=None*)

Une instance de `Cmd` ou d'une classe en héritant est un *framework* orienté ligne de commande. Il n'y a pas de bonne raison d'instancier `Cmd` directement. Elle est plutôt utile en tant que classe mère d'une classe-interprète que vous définirez afin d'hériter des méthodes de `Cmd` et d'encapsuler les opérations.

L'argument facultatif *completekey* est le nom *readline* d'une touche de complétion. Si *completekey* ne vaut pas `None` et que *readline* est disponible, la complétion de commandes est faite automatiquement.

Les arguments facultatifs *stdin* et *stdout* spécifient les objets-fichiers de lecture et d'écriture que l'instance de `Cmd` ou d'une classe fille utilisera comme entrée et sortie. Si ces arguments ne sont pas spécifiés, ils prendront comme valeur par défaut `sys.stdin` et `sys.stdout`.

Si vous souhaitez qu'un *stdin* donné soit utilisé, assurez-vous que l'attribut *use_rawinput* de l'instance vaille `False`, faute de quoi *stdin* sera ignoré.

24.2.1 Objets Cmd

Une instance de `Cmd` possède les méthodes suivantes :

`Cmd.cmdloop` (*intro=None*)

Affiche une invite de commande de manière répétée, accepte une entrée, soustrait un préfixe initial de l'entrée reçue et envoie aux méthodes d'opération la partie restante de l'entrée reçue.

L'argument facultatif est une bannière ou chaîne de caractères d'introduction à afficher avant la première invite de commande (il redéfinit l'attribut de classe *intro*).

Si le module *readline* est chargé, l'entrée héritera automatiquement d'une édition d'historique similaire à **bash** (Par exemple, `Control-P` reviendra à la dernière commande, `Control-N` avancera à la suivante, `Control-F` déplace le curseur vers la droite, `Control-B` déplace le curseur vers la gauche, etc...).

Une caractère de fin de fichier est transmis via la chaîne de caractères `'EOF'`.

Une instance d'un interpréteur reconnaîtra un nom de commande `foo` si et seulement si celle-ci possède une méthode `do_foo()`. Les lignes commençant par le caractère `'?'` sont un cas particulier : elles sont envoyées à la

méthode `do_help()`. Les lignes commençant par le caractère `'!'` sont également un cas particulier : elles sont envoyées à la méthode `do_shell()` (si une telle méthode est définie).

Cette méthode ne s'arrêtera que lorsque `postcmd()` renverra une valeur vraie. L'argument `stop` de `postcmd()` est la valeur de retour de la méthode `do_*()` correspondant à la commande.

Si la complétion est activée, la complétion de commandes sera faite automatiquement ; et la complétion d'arguments sera faite en appelant `complete_foo()` avec les arguments `text`, `line`, `begidx`, et `endidx`. `text` est le préfixe que nous cherchons à faire coïncider : toutes les valeurs renvoyées doivent commencer par ce préfixe. `line` est la ligne d'entrée actuelle sans les espaces blancs de début. `begidx` et `endidx` sont les index de début et de fin du préfixe, ils pourraient être utilisés pour fournir différentes complétions en fonction de la position de l'argument.

Toutes les classes filles de `Cmd` héritent d'une méthode `do_help()` prédéfinie. Cette méthode appellera la méthode `help_bar()` lorsqu'elle est appelée avec un argument `'bar'`. Si celle-ci n'est pas définie, elle affichera la `docstring` de `do_bar()`, (si elle a une `docstring`). Sans argument, `do_help()` listera tous les sujets d'aide (c'est à dire, toutes les commandes avec une méthode `help_*` correspondante ou commande ayant une `docstring`, elle lisera aussi les commandes non documentées.

`Cmd.onecmd(str)`

Interprète l'argument comme si il avait été entré en réponse à l'invite de commande. Cette méthode peut être surchargée, mais ne devrait normalement pas avoir besoin de l'être ; voir les méthodes `precmd()` et `postcmd()` pour altérer l'exécution d'une commande. La valeur de retour est un *flag* indiquant si l'interprétation de commandes par l'interpréteur devrait arrêter. S'il existe une méthode `do_*()` pour la commande `str`, la valeur de retour de cette méthode est renvoyée. Dans le cas contraire, la valeur de retour de la méthode `default()` est renvoyée.

`Cmd.emptyline()`

Méthode appelée quand une ligne vide est entrée en réponse à l'invite de commande. Si cette méthode n'est pas surchargée, elle répète la dernière commande non-vide entrée.

`Cmd.default(line)`

Méthode appelée lorsque le préfixe de commande d'une ligne entrée n'est pas reconnu. Si cette méthode n'est pas surchargée, elle affiche un message d'erreur et s'arrête.

`Cmd.completedefault(text, line, begidx, endidx)`

Méthode appelée pour compléter une ligne entrée quand aucune méthode `complete_*` spécifique à la commande n'est disponible. Par défaut, elle renvoie une liste vide.

`Cmd.precmd(line)`

Méthode de rappel exécutée juste avant que la ligne de commande `line` ne soit interprétée, mais après que l'invite de commande ait été généré et affiché. Cette méthode existe afin d'être surchargée par des classes filles de `Cmd`. La valeur de retour est utilisée comme la commande qui sera exécutée par la méthode `onecmd()`. L'implémentation de `precmd()` peut réécrire la commande ou simplement renvoyer `line` sans modification.

`Cmd.postcmd(stop, line)`

Méthode de rappel exécutée juste après qu'une commande ait été exécutée. Cette méthode existe afin d'être surchargée par des classes filles de `Cmd`. *line est la ligne de commande ayant été exécutée et *stop est un flag* indiquant si l'exécution sera terminée après un appel à `postcmd()`. `stop` sera la valeur de retour de `onecmd()`. La valeur de retour de cette méthode sera utilisée comme nouvelle valeur pour le *flag* interne correspondant à `stop`. Renvoyer `False` permettra à l'interprétation de continuer.

`Cmd.preloop()`

Méthode de rappel exécutée une fois lorsque `cmdloop()` est appelée. Cette méthode existe afin d'être surchargée par des classes filles de `Cmd`.

`Cmd.postloop()`

Méthode de rappel exécutée une fois lorsque `cmdloop()` va s'arrêter. Cette méthode existe afin d'être surchargée par des classes filles de `Cmd`.

Les instances de classes filles de `Cmd` possèdent des variables d'instance publiques :

`Cmd.prompt`

L'invite de commande affiché pour solliciter une entrée.

Cmd.identchars

La chaîne de caractères acceptée en tant que préfixe de commande.

Cmd.lastcmd

Le dernier préfixe de commande non-vidé vu.

Cmd.cmdqueue

Une liste de lignes entrées en file. La liste *cmdqueue* est vérifiée dans *cmdloop()* lorsqu'une nouvelle entrée est nécessitée ; si elle n'est pas vide, ses éléments seront traités dans l'ordre, comme si ils avaient entrés dans l'invite de commande.

Cmd.intro

Une chaîne de caractères à afficher en introduction ou bannière. Peut être surchargée en passant un argument à la méthode *cmdloop()*.

Cmd.doc_header

L'en-tête à afficher si la sortie de l'aide possède une section pour les commandes documentées.

Cmd.misc_header

L'en-tête à afficher si la sortie de l'aide possède une section pour divers sujets (c'est-à-dire qu'il existe des méthodes *help_*()* sans méthodes *do_*()* correspondantes).

Cmd.undoc_header

L'en-tête à afficher si la sortie de l'aide possède une section pour les commandes non documentées (c'est-à-dire qu'il existe des méthodes *do_*()* sans méthodes *help_*()* correspondantes).

Cmd.ruler

Le caractère utilisé pour afficher des lignes de séparation sous les en-têtes de messages d'aide. Si il est vide, aucune ligne de séparation n'est affichée. Par défaut, ce caractère vaut '='.

Cmd.use_rawinput

Un *flag*, valant *True* par défaut. Si ce *flag* est vrai, *cmdloop()* utilise *input()* pour afficher une invite de commande et lire la prochaine commande ; si il est faux, *sys.stdout.write()* et *sys.stdin.readline()* sont utilisées. (Cela signifie qu'en important *readline* sur les systèmes qui le supportent, l'interpréteur va automatiquement supporter une édition de ligne similaire à **Emacs** ainsi que des touches d'historique de commande).

24.2.2 Exemple

Le module *cmd* est utile pour produire des invites de commande permettant à l'utilisateur de travailler avec un programme de manière interactive.

Cette section présente un exemple simple de comment produire une invite de commande autour de quelques commandes du module *turtle*.

Des commandes *turtle* basiques telles que *forward()* sont ajoutées à une classe fille de *Cmd* avec la méthode appelée *do_forward()*. L'argument est converti en nombre et envoyé au module *turtle*. La *docstring* est utilisée dans l'utilitaire d'aide fourni par l'invite de commande.

L'exemple inclut également un utilitaire d'enregistrement et de *playback* implémenté avec la méthode *precmd()*, qui est responsable du passage de l'entrée en minuscules ainsi que d'écrire les commandes dans un fichier. La méthode *do_playback()* lit le fichier et ajoute les commandes enregistrées à *cmdqueue* pour être rejouées immédiatement :

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.  Type help or ? to list commands.\n'
    prompt = '(turtle) '

```

(suite sur la page suivante)

(suite de la page précédente)

```

file = None

# ----- basic turtle commands -----
def do_forward(self, arg):
    'Move the turtle forward by the specified distance: FORWARD 10'
    forward(*parse(arg))
def do_right(self, arg):
    'Turn turtle right by given number of degrees: RIGHT 20'
    right(*parse(arg))
def do_left(self, arg):
    'Turn turtle left by given number of degrees: LEFT 90'
    left(*parse(arg))
def do_goto(self, arg):
    'Move turtle to an absolute position with changing orientation. GOTO 100 200'
    goto(*parse(arg))
def do_home(self, arg):
    'Return turtle to the home position: HOME'
    home()
def do_circle(self, arg):
    'Draw circle with given radius an options extent and steps: CIRCLE 50'
    circle(*parse(arg))
def do_position(self, arg):
    'Print the current turtle position: POSITION'
    print('Current position is %d %d\n' % position())
def do_heading(self, arg):
    'Print the current turtle heading in degrees: HEADING'
    print('Current heading is %d\n' % (heading(),))
def do_color(self, arg):
    'Set the color: COLOR BLUE'
    color(arg.lower())
def do_undo(self, arg):
    'Undo (repeatedly) the last turtle action(s): UNDO'
def do_reset(self, arg):
    'Clear the screen and return turtle to center: RESET'
    reset()
def do_bye(self, arg):
    'Stop recording, close the turtle window, and exit: BYE'
    print('Thank you for using Turtle')
    self.close()
    bye()
    return True

# ----- record and playback -----
def do_record(self, arg):
    'Save future commands to filename: RECORD rose.cmd'
    self.file = open(arg, 'w')
def do_playback(self, arg):
    'Playback commands from a file: PLAYBACK rose.cmd'
    self.close()
    with open(arg) as f:
        self.cmdqueue.extend(f.read().splitlines())
def precmd(self, line):
    line = line.lower()
    if self.file and 'playback' not in line:
        print(line, file=self.file)
    return line
def close(self):

```

(suite sur la page suivante)

(suite de la page précédente)

```

    if self.file:
        self.file.close()
        self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()

```

Voici une session d'exemple avec l'invite de commande *turtle*. Elle montre les fonctions d'aide, utilise les lignes vides pour répéter des commandes et montre l'utilitaire de *playback* :

```

Welcome to the turtle shell.  Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
=====
bye      color      goto      home      playback  record    right
circle   forward  heading  left      position  reset     undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400

```

(suite sur la page suivante)

(suite de la page précédente)

```
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

24.3 shlex — Simple lexical analysis

Code source : [Lib/shlex.py](#)

The `shlex` class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The `shlex` module defines the following functions :

`shlex.split(s, comments=False, posix=True)`

Split the string `s` using shell-like syntax. If `comments` is `False` (the default), the parsing of comments in the given string will be disabled (setting the `commenters` attribute of the `shlex` instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the `posix` argument is false.

Note : Since the `split()` function instantiates a `shlex` instance, passing `None` for `s` will read the string to split from standard input.

`shlex.quote(s)`

Return a shell-escaped version of the string `s`. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

This idiom would be unsafe :

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

`quote()` lets you plug the security hole :

```
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l \''somefile; rm -rf ~\''
```


The quoting is compatible with UNIX shells and with `split()` :

```
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

Nouveau dans la version 3.3.

The `shlex` module defines the following class :

class `shlex.shlex` (*instream=None, infile=None, posix=False, punctuation_chars=False*)

A `shlex` instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with `read()` and `readline()` methods, or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the `infile` attribute. If the `instream` argument is omitted or equal to `sys.stdin`, this second argument defaults to « `stdin` ». The `posix` argument defines the operational mode : when `posix` is not true (default), the `shlex` instance will operate in compatibility mode. When operating in POSIX mode, `shlex` will try to be as close as possible to the POSIX shell parsing rules. The `punctuation_chars` argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values : the default value, `False`, preserves the behaviour seen under Python 3.5 and earlier. If set to `True`, then parsing of the characters `() ; <> | &` is changed : any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the `wordchars` attribute that appear in `punctuation_chars` will be removed from `wordchars`. See *Improved Compatibility with Shells* for more information.

Modifié dans la version 3.6 : The `punctuation_chars` parameter was added.

Voir aussi :

Module `configparser` Parser for configuration files similar to the Windows `.ini` files.

24.3.1 shlex Objects

A `shlex` instance has the following methods :

`shlex.get_token()`

Return a token. If tokens have been stacked using `push_token()`, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, `eof` is returned (the empty string `('')` in non-POSIX mode, and `None` in POSIX mode).

`shlex.push_token(str)`

Push the argument onto the token stack.

`shlex.read_token()`

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

`shlex.sourcehook(filename)`

When `shlex` detects a source request (see `source` below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with `open()` called on it to yield the second component. (Note : this is the reverse of the order of arguments in instance initialization !)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding “close” hook, but a `shlex` instance will call the `close()` method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the `push_source()` and `pop_source()` methods.

`shlex.push_source(newstream, newfile=None)`

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the `sourcehook()` method.

`shlex.pop_source()`

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

`shlex.error_leader(infile=None, lineno=None)`

This method generates an error message leader in the format of a Unix C compiler error label; the format is `'%s', line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage `shlex` users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of `shlex` subclasses have some public instance variables which either control lexical analysis or can be used for debugging :

`shlex.commenters`

The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

`shlex.wordchars`

The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumeric characters and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If `punctuation_chars` is not empty, the characters `~-./*?=&`, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in `punctuation_chars` will be removed from `wordchars` if they are present there.

`shlex.whitespace`

Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

`shlex.escape`

Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

`shlex.quotes`

Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

`shlex.escapedquotes`

Characters in `quotes` that will interpret escape characters defined in `escape`. This is only used in POSIX mode, and includes just `'\"'` by default.

`shlex.whitespace_split`

If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with `shlex`, getting tokens in a similar way to shell arguments. If this attribute is `True`, `punctuation_chars` will have no effect, and splitting will happen only on whitespaces. When using `punctuation_chars`, which is intended to provide parsing closer to that implemented by shells, it is advisable to leave `whitespace_split` as `False` (the default value).

shlex.infile

The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.instream

The input stream from which this *shlex* instance is reading characters.

shlex.source

This attribute is `None` by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the `source` keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the `close()` method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.debug

If this attribute is numeric and 1 or more, a *shlex* instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.lineno

Source line number (count of newlines seen so far plus one).

shlex.token

The token buffer. It may be useful to examine this when catching exceptions.

shlex.eof

Token used to determine end of file. This will be set to the empty string (`' '`), in non-POSIX mode, and to `None` in POSIX mode.

shlex.punctuation_chars

Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed : for example, “>>” could be returned as a token, even though it may not be recognised as such by shells.

Nouveau dans la version 3.6.

24.3.2 Parsing Rules

When operating in non-POSIX mode, *shlex* will try to obey to the following rules.

- Quote characters are not recognized within words (`Do"Not"Separate` is parsed as the single word `Do"Not"Separate`);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words (`"Do"Separate` is parsed as `"Do"` and `Separate`);
- If `whitespace_split` is `False`, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is `True`, *shlex* will only split words in whitespaces;
- EOF is signaled with an empty string (`' '`);
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, *shlex* will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words (`"Do"Not"Separate"` is parsed as the single word `DoNotSeparate`);
- Non-quoted escape characters (e.g. `'\ '`) preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of *escapedquotes* (e.g. `" ' "`) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of *escapedquotes* (e.g. `' " '`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in *escape*. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.

- EOF is signaled with a *None* value;
- Quoted empty strings (' ') are allowed.

24.3.3 Improved Compatibility with Shells

Nouveau dans la version 3.6.

The *shlex* class provides compatibility with the parsing performed by common Unix shells like *bash*, *dash*, and *sh*. To take advantage of this compatibility, specify the *punctuation_chars* argument in the constructor. This defaults to *False*, which preserves pre-3.6 behaviour. However, if it is set to *True*, then parsing of the characters *() ; <> | &* is changed : any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet :

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> list(shlex.shlex(text))
['a', '&', '&', 'b', ';', 'c', '&', '&', 'd', '||', '|', '|', 'e', ';', 'f', '>',
"'abc'", ';', '(', 'def', "\"ghi\"", ')']
>>> list(shlex.shlex(text, punctuation_chars=True))
['a', '&&', 'b', ';', 'c', '&&', 'd', '|||', 'e', ';', 'f', '>', "'abc'",
';', '(', 'def', "\"ghi\"", ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing *True* as the value for the *punctuation_chars* parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example :

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

Note : When *punctuation_chars* is specified, the *wordchars* attribute is augmented with the characters *~-./*? =*. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. *--color=auto*). Hence :

```
>>> import shlex
>>> s = shlex.shlex('~ / a && b - c --color=auto || d *.py?',
...               punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '|||', 'd', '*.py?']
```

For best effect, *punctuation_chars* should be set in conjunction with *posix=True*. (Note that *posix=False* is the default for *shlex*.)

Interfaces Utilisateur Graphiques avec Tk

Tk/Tcl fait depuis longtemps partie intégrante de Python. Il fournit un jeu d'outils robustes et indépendants de la plateforme pour gérer des fenêtres. Disponible aux développeurs via le paquet *tkinter* et ses extensions, les modules *tkinter.tix* et *tkinter.ttk*.

Le paquet *tkinter* est une fine couche orientée objet au dessus de Tcl/Tk. Pour utiliser le module *tkinter*, vous n'avez pas à écrire de code Tcl, mais vous devrez consulter la documentation de Tk, et parfois la documentation de Tcl. Le module *tkinter* est un ensemble de surcouches implémentant les *widgets* Tk en classes Python. De plus, le module interne *_tkinter* fournit un mécanisme robuste permettant à des fils d'exécution Python et Tcl d'interagir.

Les avantages de *tkinter* sont sa rapidité, et qu'il est généralement fourni nativement avec Python. Bien que sa documentation soit mauvaise, d'autres ressources existent : des références, tutoriels, livres, ... Le module *tkinter* est aussi célèbre pour son aspect vieillot, cependant il a été grandement amélioré depuis Tk 8.5. Néanmoins, il existe bien d'autres bibliothèques d'interfaces graphiques qui pourraient vous intéresser. Pour plus d'informations sur les alternatives, consultez le chapitre *Autres paquets d'interface graphique utilisateur*.

25.1 *tkinter* — Interface Python pour Tcl/Tk

Code source : `Lib/tkinter/__init__.py`

Le paquet *tkinter* (« interface Tk ») est l'interface Python standard de la boîte à outils d'IUG Tk. Tk et *tkinter* sont disponibles sur la plupart des plates-formes Unix, ainsi que sur les systèmes Windows (Tk lui-même ne fait pas partie de Python ; il est maintenu par ActiveState).

Exécuter `python -m tkinter` depuis la ligne de commande ouvre une fenêtre de démonstration d'une interface Tk simple, vous indiquant que *tkinter* est correctement installé sur votre système et indiquant également quelle version de Tcl/Tk est installée ; vous pouvez donc lire la documentation Tcl/Tk spécifique à cette version.

Voir aussi :

Documentation de Tkinter :

Python Tkinter Resources Le *Python Tkinter Topic Guide* fournit beaucoup d'informations sur l'utilisation de Tk à partir de Python et des liens vers d'autres sources d'information sur Tk.

TKDocs Tutoriel complet plus convivial pour certains des objets graphiques.

Tkinter 8.5 reference : a GUI for Python Documents de référence en ligne.

Documents Tkinter sur effbot Référence en ligne pour *tkinter* réalisée par *effbot.org*.

Programming Python Livre de Mark Lutz, qui couvre excellemment bien Tkinter.

Modern Tkinter for Busy Python Developers Book by Mark Roseman about building attractive and modern graphical user interfaces with Python and Tkinter.

Python and Tkinter Programming Livre de John Grayson (ISBN 1-884777-81-3).

Documentation de Tcl/Tk :

Commandes Tk La plupart des commandes sont disponibles sous forme de classes *tkinter* ou *tkinter.ttk*. Modifiez “8.6” pour correspondre à votre version installée de Tcl/Tk.

Pages de manuel récentes de Tcl/Tk Manuels récents de Tcl/Tk sur *www.tcl.tk*.

ActiveState Tcl Home Page Le développement de Tk/Tcl se déroule en grande partie au sein d’ActiveState.

Tcl and the Tk Toolkit Livre de John Ousterhout, l’inventeur de Tcl.

Practical Programming in Tcl and Tk Le livre encyclopédique de Brent Welch.

25.1.1 Modules Tkinter

La plupart du temps, *tkinter* est tout ce dont vous avez vraiment besoin mais un certain nombre de modules supplémentaires sont également disponibles. L’interface Tk est située dans un module binaire nommé *_tkinter*. Ce module contient l’interface de bas niveau vers Tk et ne doit jamais être utilisé directement par les développeurs. Il s’agit généralement d’une bibliothèque partagée (ou DLL) mais elle peut, dans certains cas, être liée statiquement à l’interpréteur Python.

En plus du module d’interface Tk, *tkinter* inclut un certain nombre de modules Python, *tkinter.constants* étant l’un des plus importants. Importer *tkinter* charge automatiquement *tkinter.constants* donc, habituellement, pour utiliser Tkinter, tout ce dont vous avez besoin est une simple commande d’importation :

```
import tkinter
```

Ou, plus souvent :

```
from tkinter import *
```

class *tkinter.Tk* (*screenName=None, baseName=None, className='Tk', useTk=1*)

La classe *Tk* est instanciée sans argument. Cela crée un widget de haut niveau de Tk qui est généralement la fenêtre principale d’une application. Chaque instance a son propre interpréteur Tcl associé.

tkinter.Tcl (*screenName=None, baseName=None, className='Tk', useTk=0*)

La fonction *Tcl()* est une fonction fabrique qui crée un objet similaire à celui créé par la classe *Tk*, sauf qu’elle n’initialise pas le sous-système Tk. Ceci est le plus souvent utile lorsque vous pilotez l’interpréteur Tcl dans un environnement où vous ne voulez pas créer des fenêtres de haut niveau supplémentaires, ou alors si c’est impossible (comme les systèmes Unix/Linux sans un serveur X). Un objet créé par *Tcl()* peut avoir une fenêtre de haut niveau créée (et le sous-système Tk initialisé) en appelant sa méthode *loadtk()*.

Parmi les modules qui savent gérer Tk, nous pouvons citer :

tkinter.scrolledtext Outil d’affichage de texte avec une barre de défilement verticale intégrée.

tkinter.colorchooser Boîte de dialogue permettant à l’utilisateur de choisir une couleur.

tkinter.commondialog Classe de base pour les boîtes de dialogue définies dans les autres modules listés ici.

tkinter.filedialog Boîtes de dialogue standard permettant à l’utilisateur de spécifier un fichier à ouvrir ou à enregistrer.

tkinter.font Utilitaires pour gérer les polices de caractères.

tkinter.messagebox Accès aux boîtes de dialogue Tk standard.

tkinter.simpledialog Boîtes de dialogue simples et fonctions utilitaires.

tkinter.dnd Support du glisser-déposer pour *tkinter*. Il s'agit d'une méthode expérimentale qui ne sera plus maintenue lorsqu'elle sera remplacée par Tk DND.

turtle Tortue graphique dans une fenêtre Tk.

25.1.2 Guide de survie Tkinter

Cette section n'est pas conçue pour être un tutoriel exhaustif de Tk ou Tkinter. Il s'agit plutôt d'un guide d'introduction au système.

Crédits :

- *Tk* a été écrit par John Ousterhout de Berkeley.
- *Tkinter* a été écrit par Steen Lumholt et Guido van Rossum.
- Ce guide de survie a été écrit par Matt Conway de l'Université de Virginie.
- Le rendu HTML, avec quelques modifications, a été réalisé à partir d'une version FrameMaker par Ken Manheimer.
- Fredrik Lundh a élaboré et mis à jour les descriptions de l'interface des classes, en cohérence avec Tk 4.2.
- Mike Clarkson a converti la documentation en LaTeX et a compilé le chapitre *Interface utilisateur* du manuel de référence.

Mode d'emploi

Cette section est divisée en deux parties : la première moitié (à peu près) couvre la partie théorique, tandis que la seconde moitié peut être utilisée comme guide pratique.

Lorsque l'on essaie de répondre à des questions sur la manière de faire « ceci ou cela », il est souvent préférable de trouver comment le faire en Tk, puis de le convertir en fonction correspondante *tkinter*. Les programmeurs Python peuvent souvent deviner la commande Python correcte en consultant la documentation Tk. Cela signifie que pour utiliser Tkinter, vous devez en savoir un peu plus sur Tk. Ce document ne peut pas remplir ce rôle, alors le mieux que nous puissions faire est de vous indiquer la meilleure documentation qui existe. Voici quelques conseils :

- Les auteurs conseillent fortement d'obtenir une copie des pages de manuel de Tk. En particulier, les pages de manuel dans le répertoire `manN` sont les plus utiles. Les pages de manuel `man3` décrivent l'interface C de la bibliothèque Tk et ne sont donc pas particulièrement utiles aux développeurs de scripts.
- Addison-Wesley a publié un livre intitulé *Tcl and the Tk Toolkit* de John Ousterhout (ISBN 0-201-63337-X) qui est une bonne introduction à Tcl et Tk pour débutants. Le livre n'est pas exhaustif et, pour beaucoup de détails, il renvoie aux pages du manuel.
- `tkinter/__init__.py` est souvent un dernier recours, mais peut être un bon endroit où aller quand rien d'autre ne fait sens.

Un simple programme *Hello World*

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()
```

(suite sur la page suivante)

(suite de la page précédente)

```

def create_widgets(self):
    self.hi_there = tk.Button(self)
    self.hi_there["text"] = "Hello World\n(click me)"
    self.hi_there["command"] = self.say_hi
    self.hi_there.pack(side="top")

    self.quit = tk.Button(self, text="QUIT", fg="red",
                           command=self.master.destroy)
    self.quit.pack(side="bottom")

def say_hi(self):
    print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()

```

25.1.3 Un (très) rapide aperçu de Tcl/Tk

La hiérarchie de classes semble compliquée mais, dans la pratique, les développeurs d'applications se réfèrent presque toujours aux classes situées tout en bas de la hiérarchie.

Notes :

- Ces classes sont fournies dans le but d'organiser certaines fonctions sous un seul espace de nommage. Elles n'ont pas vocation à être instanciées indépendamment.
- La classe `Tk` est destinée à être instanciée une seule fois dans une application. Les développeurs d'applications n'ont pas besoin d'en instancier une explicitement, Le système en crée une au besoin quand une des autres classes est instanciée.
- La classe `Widget` n'est pas destinée à être instanciée, elle est destinée uniquement au sous-classement pour faire de « vrais » objets graphiques (en C++, on appelle cela une « classe abstraite »).

Pour comprendre cette documentation, il y aura des moments où vous aurez besoin de savoir comment lire de courts passages de Tk et comment identifier les différentes parties d'une commande Tk. (Voir la section [Correspondance entre Basic Tk et Tkinter](#) pour les équivalents `tkinter` de ce qui suit).

Les scripts Tk sont des programmes Tcl. Comme tous les programmes Tcl, les scripts Tk ne sont que des listes de commandes séparées par des espaces. Un objet graphique Tk n'est constitué que de sa *classe*, des *options* qui l'aident à se configurer et des *actions* qui lui font faire des choses utiles.

Pour créer un objet graphique en Tk, la commande est toujours de la forme :

```
classCommand newPathname options
```

classCommand indique le type d'objet graphique à réaliser (un bouton, une étiquette, un menu...)

newPathname est le nouveau nom pour cet objet graphique. Tous les noms dans Tk doivent être uniques. Pour vous aider à respecter cette règle, les objets graphiques dans Tk sont nommés avec des *noms d'accès*, tout comme les fichiers dans le système de fichiers. L'objet graphique de niveau supérieur, la racine (*root* en anglais), s'appelle `.` (point) et les enfants sont délimités par plusieurs points. Par exemple, `.myApp.controlPanel.okButton` pourrait être le nom d'un objet graphique.

options configure l'apparence de l'objet graphique et, dans certains cas, son comportement. Les options se présentent sous la forme d'une liste de paramètres et de valeurs. Les paramètres sont précédés d'un « - », comme les paramètres d'une ligne de commande du shell Unix, et les valeurs sont mises entre guillemets si elles font plus d'un mot.

Par exemple :

```
button .fred -fg red -text "hi there"
  ^      ^      \_____/
  |      |      |
class   new      options
command widget (-opt val -opt val ...)
```

Une fois créé, le chemin d'accès à l'objet graphique devient une nouvelle commande. Cette nouvelle *commande d'objet graphique* est l'interface du programmeur pour que le nouvel objet graphique effectue une *action*. En C, cela prend la forme `someAction(fred, someOptions)`, en C++, cela prend la forme `fred.someAction(someOptions)` et, en Tk, vous dites :

```
.fred someAction someOptions
```

Notez que le nom de l'objet, `.fred`, commence par un point.

Comme vous pouvez vous y attendre, les valeurs autorisées pour *someAction* dépendent de la classe de l'objet graphique : `.fred disable` fonctionne si `fred` est un bouton (`fred` devient grisé), mais ne fonctionne pas si `fred` est une étiquette (la désactivation des étiquettes n'existe pas dans Tk).

Les valeurs possibles de *someOptions* dépendent de l'action. Certaines actions, comme `disable`, ne nécessitent aucun argument ; d'autres, comme la commande `delete` d'une zone de saisie, nécessitent des arguments pour spécifier l'étendue du texte à supprimer.

25.1.4 Correspondance entre *Basic Tk* et *Tkinter*

Les commandes de classes dans Tk correspondent aux constructeurs de classes dans Tkinter.

```
button .fred          =====> fred = Button()
```

Le constructeur d'un objet est implicite dans le nouveau nom qui lui est donné lors de la création. Dans Tkinter, les constructeurs sont spécifiés explicitement.

```
button .panel.fred    =====> fred = Button(panel)
```

Les options de configuration dans Tk sont données dans des listes de paramètres séparés par des traits d'union suivies de leurs valeurs. Dans Tkinter, les options sont spécifiées sous forme d'arguments par mots-clés dans le constructeur d'instance, et d'arguments par mots-clés pour configurer les appels ou sous forme d'une entrée, dans le style dictionnaire, d'instance pour les instances établies. Voir la section *Définition des options* pour la façon de définir les options.

```
button .fred -fg red    =====> fred = Button(panel, fg="red")
.fred configure -fg red  =====> fred["fg"] = red
OR ==> fred.config(fg="red")
```

Dans Tk, pour effectuer une action sur un objet graphique, utilisez le nom de l'objet graphique comme une commande et faites-le suivre d'un nom d'action, éventuellement avec des arguments (options). Dans Tkinter, vous appelez des méthodes sur l'instance de classe pour invoquer des actions sur l'objet graphique. Les actions (méthodes) qu'un objet graphique donné peut effectuer sont listées dans `tkinter/__init__.py`.

```
.fred invoke          =====> fred.invoke()
```

Pour donner un objet graphique à l'empaqueteur (ce qui gère la disposition à l'écran), appelez `pack` avec des arguments optionnels. Dans Tkinter, la classe `Pack` contient toutes ces fonctionnalités et les différentes formes de la commande `pack` sont implémentées comme méthodes. Tous les objets graphiques de *tkinter* sont sous-classés depuis l'empaqueteur et donc héritent de toutes les méthodes d'empaquetage. Voir la documentation du module *tkinter.tix* pour plus d'informations sur le gestionnaire de disposition des formulaires.

```
pack .fred -side left          =====> fred.pack(side="left")
```

25.1.5 Relations entre Tk et Tkinter

De haut en bas :

Votre application (Python) Une application Python fait un appel *tkinter*.

tkinter (paquet Python) Cet appel (par exemple, la création d'un objet graphique de type bouton) est implémenté dans le paquet *tkinter*, qui est écrit en Python. Cette fonction Python analyse les commandes et les arguments et les convertit en une forme qui les fait ressembler à un script Tk au lieu d'un script Python.

_tkinter (C) Ces commandes et leurs arguments sont passés à une fonction C dans le module d'extension *_tkinter* — notez le tiret bas.

Objets graphiques Tk (C et Tcl) Cette fonction C est capable d'effectuer des appels vers d'autres modules C, y compris les fonctions C qui composent la bibliothèque Tk. Tk est implémenté en C et un peu en Tcl. La partie Tcl des objets graphiques Tk est utilisée pour lier certains comportements par défaut aux objets graphiques, et est exécutée une fois au moment où le paquet Python *tkinter* est importé (cette étape est transparente pour l'utilisateur).

Tk (C) La partie Tk des objets graphiques Tk implémente la correspondance finale avec ...

Xlib (C) la bibliothèque *Xlib* pour dessiner des éléments graphiques à l'écran.

25.1.6 Guide pratique

Définition des options

Les options contrôlent des paramètres tels que la couleur et la largeur de la bordure d'un objet graphique. Les options peuvent être réglées de trois façons :

Lors de la création de l'objet, à l'aide d'arguments par mots-clés

```
fred = Button(self, fg="red", bg="blue")
```

Après la création de l'objet, en manipulant le nom de l'option comme une entrée de dictionnaire

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

Utilisez la méthode `config()` pour mettre à jour plusieurs attributs après la création de l'objet

```
fred.config(fg="red", bg="blue")
```

Pour l'explication complète d'une option donnée et de son comportement, voir les pages de manuel Tk de l'objet graphique en question.

Notez que les pages de manuel listent « OPTIONS STANDARD » et « OPTIONS SPÉCIFIQUES D'OBJETS GRAPHIQUES » pour chaque objet graphique. La première est une liste d'options communes à de nombreux objets graphiques, la seconde est une liste d'options propres à cet objet graphique particulier. Les options standard sont documentées sur la page de manuel *options(3)*.

Aucune distinction n'est faite dans ce document entre les options standard et les options spécifiques à un objet graphique. Certaines options ne s'appliquent pas à certains types d'objets graphiques. La réaction d'un objet graphique donné à une option particulière dépend de la classe de l'objet graphique ; les boutons possèdent une option `command`, pas les étiquettes.

Les options gérées par un objet graphique donné sont listées dans la page de manuel de cet objet graphique, ou peuvent être interrogées à l'exécution en appelant la méthode `config()` sans argument, ou en appelant la méthode `keys()` sur

cet objet graphique. La valeur de retour de ces appels est un dictionnaire dont la clé est le nom de l'option sous forme de chaîne (par exemple, 'relief') et dont les valeurs sont des *5-uplets*.

Certaines options, comme `bg`, sont des synonymes d'options communes qui ont des noms longs (`bg` est une abréviation pour `background` « arrière-plan »). Passer le nom abrégé d'une option à la méthode `config()` renvoie un couple, pas un quintuplet. Le couple renvoyé contient le nom abrégé et le nom *réel* de l'option, par exemple ('bg', 'background').

Index	Signification	Exemple
0	nom des options	'relief'
1	nom de l'option pour la recherche dans la base de données	'relief'
2	classe de l'option pour la recherche dans la base de données	'Relief'
3	valeur par défaut	'raised'
4	valeur actuelle	'groove'

Exemple :

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Bien sûr, le dictionnaire affiché contient toutes les options disponibles et leurs valeurs. Ceci n'est donné qu'à titre d'exemple.

L'empaqueteur

L'empaqueteur est l'un des mécanismes de Tk pour la gestion de la disposition des éléments sur l'écran. Les gestionnaires de géométrie sont utilisés pour spécifier le positionnement relatif du positionnement des objets graphiques dans leur conteneur — leur *constructeur* mutuel. Contrairement au plus encombrant *placeur* (qui est utilisé moins souvent, et nous n'en parlons pas ici), l'empaqueteur prend les spécifications qualitatives de relation — *above*, *to the left of*, *filling*, etc — et calcule tout pour déterminer les coordonnées exactes du placement pour vous.

La taille d'un objet graphique *constructeur* est déterminée par la taille des « objets graphiques hérités » à l'intérieur. L'empaqueteur est utilisé pour contrôler l'endroit où les objets graphiques hérités apparaissent à l'intérieur du constructeur dans lequel ils sont empaquetés. Vous pouvez regrouper des objets graphiques dans des cadres, et des cadres dans d'autres cadres, afin d'obtenir le type de mise en page souhaité. De plus, l'arrangement est ajusté dynamiquement pour s'adapter aux changements incrémentiels de la configuration, une fois qu'elle est empaquetée.

Notez que les objets graphiques n'apparaissent pas tant que leur disposition n'a pas été spécifiée avec un gestionnaire de géométrie. C'est une erreur de débutant courante de ne pas tenir compte de la spécification de la géométrie, puis d'être surpris lorsque l'objet graphique est créé mais que rien n'apparaît. Un objet graphique n'apparaît qu'après que, par exemple, la méthode `pack()` de l'empaqueteur lui ait été appliquée.

La méthode `pack()` peut être appelée avec des paires mot-clé-option/valeur qui contrôlent où l'objet graphique doit apparaître dans son conteneur et comment il doit se comporter lorsque la fenêtre principale de l'application est redimensionnée. En voici quelques exemples :

```
fred.pack()                                # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

Options de l'empaqueteur

Pour de plus amples informations sur l'empaqueteur et les options qu'il peut prendre, voir les pages de manuel et la page 183 du livre de John Ousterhout.

anchor Type d'ancrage. Indique l'endroit où l'empaqueteur doit placer chaque enfant dans son espace.

expand Booléen, 0 ou 1.

fill Valeurs acceptées : 'x', 'y', 'both', 'none'.

ipadx et ipady Une distance — désignant l'écart interne de chaque côté de l'objet graphique hérité.

padx et pady Une distance — désignant l'écart externe de chaque côté de l'objet graphique hérité.

side Valeurs acceptées : 'left', 'right', 'top', 'bottom'.

Association des variables de l'objet graphique

L'assignation d'une valeur à certains objets graphiques (comme les objets graphique de saisie de texte) peut être liée directement aux variables de votre application à l'aide d'options spéciales. Ces options sont `variable`, `textvariable`, `onvalue`, `offvalue` et `value`. Ce lien fonctionne dans les deux sens : si la variable change pour une raison ou pour une autre, l'objet graphique auquel elle est connectée est mis à jour pour refléter la nouvelle valeur.

Malheureusement, dans l'implémentation actuelle de `tkinter` il n'est pas possible de passer une variable Python arbitraire à un objet graphique via une option `variable` ou `textvariable`. Les seuls types de variables pour lesquels cela fonctionne sont les variables qui sont sous-classées à partir d'une classe appelée *Variable*, définie dans `tkinter`.

Il existe de nombreuses sous-classes utiles de *Variable* déjà définies : `StringVar`, `IntVar`, `DoubleVar` et `BooleanVar`. Pour lire la valeur courante d'une telle variable, appelez la méthode `get()` dessus et, pour changer sa valeur, appelez la méthode `set()`. Si vous suivez ce protocole, l'objet graphique suivra toujours la valeur de la variable, sans autre intervention de votre part.

Par exemple :

```
class App(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("hi. contents of entry is now ---->",
              self.contents.get())
```

Le gestionnaire de fenêtres

Dans Tk, il y a une commande pratique, `wm`, pour interagir avec le gestionnaire de fenêtres. Les options de la commande `wm` vous permettent de contrôler les titres, le placement, les icônes en mode *bitmap* et encore d'autres choses du même genre. Dans *tkinter*, ces commandes ont été implémentées en tant que méthodes sur la classe `Wm`. Les objets graphiques de haut niveau sont sous-classés à partir de la classe `Wm`, ils peuvent donc appeler directement les méthodes de `Wm`.

Pour accéder à la fenêtre du plus haut niveau qui contient un objet graphique donné, vous pouvez souvent simplement vous référer au parent de cet objet graphique. Bien sûr, si l'objet graphique a été empaqueté à l'intérieur d'un cadre, le parent ne représentera pas la fenêtre de plus haut niveau. Pour accéder à la fenêtre du plus haut niveau qui contient un objet graphique arbitraire, vous pouvez appeler la méthode `_root()`. Cette méthode commence par un soulignement pour indiquer que cette fonction fait partie de l'implémentation, et non d'une interface avec la fonctionnalité Tk.

Voici quelques exemples d'utilisation courante :

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Types de données des options Tk

anchor Les valeurs acceptées sont des points cardinaux : « n », « ne », « e », « se », « s », « sw », « w », « nw » et « center ».

bitmap Il y a huit bitmaps intégrés nommés : « error », « gray25 », « gray50 », « hourglass », « info », « questhead », « question », « warning ». Pour spécifier un nom de fichier bitmap X, indiquez le chemin complet du fichier, précédé de @, comme dans « @/usr/contrib/bitmaps/gumby.bit ».

boolean Vous pouvez lui donner les entiers 0 ou 1 ou les chaînes de caractères "yes" ou "no".

callback N'importe quelle fonction Python qui ne prend pas d'argument. Par exemple :

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

color Les couleurs peuvent être données sous forme de noms de couleurs Xorg dans le fichier *rgb.txt*, ou sous forme de chaînes représentant les valeurs RVB en 4 bits : « #RGB », 8 bits : « #RRVVB » , 12 bits : « #RRRVVVBBB » , ou 16 bits : « #RRRRVVVVBBBB » , où R,V,B représente ici tout chiffre hexadécimal valide. Voir page 160 du livre d'Ousterhout pour plus de détails.

cursor Les noms de curseurs Xorg standard que l'on trouve dans *cursorfont.h* peuvent être utilisés, sans le préfixe `XC_`. Par exemple pour obtenir un curseur en forme de main (`XC_hand2`), utilisez la chaîne « hand2 ». Vous pouvez également spécifier votre propre bitmap et fichier masque. Voir page 179 du livre d'Ousterhout.

distance Les distances à l'écran peuvent être spécifiées en pixels ou en distances absolues. Les pixels sont donnés sous forme de nombres et les distances absolues sous forme de chaînes de caractères, le dernier caractère indiquant les unités : *c* pour les centimètres, *i* pour les pouces (*inches* en anglais), *m* pour les millimètres, *p* pour les points d'impression. Par exemple, 3,5 pouces est noté « 3.5i ».

font Tk utilise un format de nom de police sous forme de liste, tel que {courier 10 bold}. Les tailles de polices avec des nombres positifs sont mesurées en points ; les tailles avec des nombres négatifs sont mesurées en pixels.

geometry Il s'agit d'une chaîne de caractères de la forme `largeurxhauteur`, où la largeur et la hauteur sont mesurées en pixels pour la plupart des objets graphiques (en caractères pour les objets graphiques affichant du texte). Par exemple : `fred["geometry"] = "200x100"`.

justify Les valeurs acceptées sont les chaînes de caractères : « left », « center », « right » et « fill ».

region c'est une chaîne de caractères avec quatre éléments séparés par des espaces, chacun d'eux étant une distance valide (voir ci-dessus). Par exemple : "2 3 4 5""et "" 3i 2i 4.5i 2i""et "" 3c 2c 4c 10.43c" sont toutes des régions valides.

relief Détermine le style de bordure d'un objet graphique. Les valeurs valides sont : "raised", "sunken", "flat", "groove", et "ridge".

scrollcommand C'est presque toujours la méthode `set()` d'un objet graphique de défilement, mais peut être n'importe quelle méthode d'objet graphique qui prend un seul argument.

wrap Doit être l'un d'eux : "none", "char", ou "word".

Liaisons et événements

La méthode `bind` de la commande d'objet graphique vous permet de surveiller certains événements et d'avoir un déclencheur de fonction de rappel lorsque ce type d'événement se produit. La forme de la méthode de liaison est la suivante :

```
def bind(self, sequence, func, add='')
```

où :

sequence est une chaîne de caractères qui indique le type d'événement cible. (Voir la page du manuel de `bind` et la page 201 du livre de John Ousterhout pour plus de détails).

func est une fonction Python, prenant un argument, à invoquer lorsque l'événement se produit. Une instance d'événement sera passée en argument. (Les fonctions déployées de cette façon sont communément appelées *callbacks* ou « fonctions de rappel » en français).

add est facultative, soit ' ' ou '+'. L'envoi d'une chaîne de caractères vide indique que cette liaison doit remplacer toute autre liaison à laquelle cet événement est associé. L'envoi de «+» signifie que cette fonction doit être ajoutée à la liste des fonctions liées à ce type d'événement.

Par exemple :

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

Remarquez comment on accède au champ *objet graphique* de l'événement dans la fonction de rappel `turn_red()`. Ce champ contient l'objet graphique qui a capturé l'événement Xorg. Le tableau suivant répertorie les autres champs d'événements auxquels vous pouvez accéder, et comment ils sont nommés dans Tk, ce qui peut être utile lorsque vous vous référez aux pages de manuel Tk.

Tk	Champ <i>évènement</i> de Tkinter	Tk	Champ <i>évènement</i> de Tkinter
%f	focus	%A	char
%h	hauteur	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

Le paramètre index

Un certain nombre d'objets graphiques nécessitent le passage de paramètres « indicés ». Ils sont utilisés pour pointer vers un endroit spécifique dans un objet graphique de type *Texte*, ou vers des caractères particuliers dans un objet graphique de type *Entrée*, ou vers des éléments de menu particuliers dans un objet graphique de type *Menu*.

Index des objets graphique de type *Entrée* (*index*, *view index*, etc.) Les objets graphiques de type *Entrée* ont des options qui se réfèrent à la position des caractères dans le texte affiché. Vous pouvez utiliser ces fonctions *tkinter* pour accéder à ces points spéciaux dans les objets graphiques texte :

Index des objets graphiques texte La notation de l'index des objets graphiques de type *Texte* est très riche et mieux décrite dans les pages du manuel Tk.

Index menu (*menu.invoke()*, *menu.entryconfig()*, etc.) Certaines options et méthodes pour manipuler les menus nécessitent des éléments de spécifiques. Chaque fois qu'un index de menu est nécessaire pour une option ou un paramètre, vous pouvez utiliser :

- un entier qui fait référence à la position numérique de l'entrée dans l'objet graphique, comptée à partir du haut, en commençant par 0 ;
- la chaîne de caractères "active", qui fait référence à la position du menu qui se trouve actuellement sous le curseur ;
- la chaîne de caractères "last" qui fait référence au dernier élément du menu ;
- un entier précédé de @, comme dans @6, où l'entier est interprété comme une coordonnée y de pixels dans le système de coordonnées du menu ;
- la chaîne de caractères "none", qui n'indique aucune entrée du menu, le plus souvent utilisée avec *menu.activate()* pour désactiver toutes les entrées, et enfin,
- une chaîne de texte dont le motif correspond à l'étiquette de l'entrée de menu, telle qu'elle est balayée du haut vers le bas du menu. Notez que ce type d'index est considéré après tous les autres, ce qui signifie que les correspondances pour les éléments de menu étiquetés *last*, *active* ou *none* peuvent être interprétés comme les littéraux ci-dessus, plutôt.

Images

Des images de différents formats peuvent être créées à travers la sous-classe correspondante de *tkinter.Image* :

- *BitmapImage* pour les images au format *XBM*.
- *PhotoImage* pour les images aux formats *PGM*, *PPM*, *GIF* et *PNG*. Ce dernier est géré à partir de Tk 8.6.

L'un ou l'autre type d'image est créé par l'option *file* ou *data* (d'autres options sont également disponibles).

L'objet image peut alors être utilisé partout où un objet graphique sait gérer une option *image* (par ex. étiquettes, boutons, menus). Dans ces cas, Tk ne conserve pas de référence à l'image. Lorsque la dernière référence Python à l'objet image est supprimée, les données de l'image sont également supprimées, et Tk affiche une boîte vide à l'endroit où l'image était utilisée.

Voir aussi :

Le paquet *Pillow* ajoute la prise en charge de formats tels que *BMP*, *JPEG*, *TIFF* et *WebP*, entre autres.

25.1.7 Gestionnaires de fichiers

Tk vous permet d'enregistrer et de *désenregistrer* une fonction de rappel qui est appelée depuis la boucle principale de Tk lorsque des entrées-sorties sont possibles sur un descripteur de fichier. Un seul gestionnaire peut être enregistré par descripteur de fichier. Exemple de code :

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

Cette fonction n'est pas disponible sous Windows.

Dans la mesure où vous ne savez pas combien d'octets sont disponibles en lecture, il ne faut pas utiliser les méthodes *BufferedIOBase* ou *TextIOBase* *read()* ou *readline()*, car elles requièrent d'indiquer le nombre de *bytes* à lire. Pour les connecteurs, les méthodes *recv()* ou *recvfrom()* fonctionnent bien ; pour les autres fichiers, utilisez des lectures brutes ou *os.read(file.fileno(), maxbytecount)*.

Widget.tk.createfilehandler(file, mask, func)

Enregistre la fonction de rappel du gestionnaire de fichiers *func*. L'argument *file* peut être soit un objet avec une méthode *fileno()* (comme un objet fichier ou connecteur), soit un descripteur de fichier de type entier. L'argument *mask* est une combinaison *OU* de l'une des trois constantes ci-dessous. La fonction de rappel s'utilise comme suit :

```
callback(file, mask)
```

Widget.tk.deletefilehandler(file)

Désenregistre un gestionnaire de fichiers.

tkinter.READABLE

tkinter.WRITABLE

tkinter.EXCEPTION

Constantes utilisées dans les arguments *mask*.

25.2 *tkinter.ttk* — Tk themed widgets

Source code : <Lib/tkinter/ttk.py>

The *tkinter.ttk* module provides access to the Tk themed widget set, introduced in Tk 8.5. If Python has not been compiled against Tk 8.5, this module can still be accessed if *Tile* has been installed. The former method using Tk 8.5 provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

The basic idea for *tkinter.ttk* is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

Voir aussi :

Tk Widget Styling Support A document introducing theming support for Tk

25.2.1 Using Ttk

To start using Ttk, import its module :

```
from tkinter import ttk
```

To override the basic Tk widgets, the import should follow the Tk import :

```
from tkinter import *
from tkinter.ttk import *
```

That code causes several *tkinter.ttk* widgets (Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar) to automatically replace the Tk widgets.

This has the direct benefit of using the new widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as « fg », « bg » and others related to widget styling are no longer present in Ttk widgets. Instead, use the *ttk.Style* class for improved styling effects.

Voir aussi :

Converting existing applications to use Tile widgets A monograph (using Tcl terminology) about differences typically encountered when moving applications to use the new widgets.

25.2.2 Ttk Widgets

Ttk comes with 17 widgets, eleven of which already existed in tkinter : Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar. The other six are new : *Combobox*, *Notebook*, *Progressbar*, *Separator*, *Sizegrip* and *Treeview*. And all them are subclasses of *Widget*.

Using the Ttk widgets gives the application an improved look and feel. As discussed above, there are differences in how the styling is coded.

Tk code :

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk code :

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about *TtkStyling*, see the *Style* class documentation.

25.2.3 Widget

`ttk.Widget` defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

Standard Options

All the `ttk` Widgets accepts the following options :

Option	Description
<code>classe</code>	Specifies the window class. The class is used when querying the option database for the window's other options, to determine the default bindtags for the window, and to select the widget's default layout and style. This option is read-only, and may only be specified when the window is created.
<code>cursor</code>	Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited for the parent widget.
<code>takefocus</code>	Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0 is returned, it means that the window should be skipped entirely during keyboard traversal. If 1, it means that the window should receive the input focus as long as it is viewable. And an empty string means that the traversal scripts make the decision about whether or not to focus on the window.
<code>style</code>	May be used to specify a custom widget style.

Scrollable Widget Options

The following options are supported by widgets that are controlled by a scrollbar.

Option	Description
<code>xscrollcommand</code>	Used to communicate with horizontal scrollbars. When the view in the widget's window change, the widget will generate a Tcl command based on the scrollcommand. Usually this option consists of the method <code>Scrollbar.set()</code> of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.
<code>yscrollcommand</code>	Used to communicate with vertical scrollbars. For some more information, see above.

Label Options

The following options are supported by labels, buttons and other button-like widgets.

Option	Description
text	Specifies a text string to be displayed inside the widget.
textvariable	Specifies a name whose value will be used in place of the text option resource.
underline	If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.
image	Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list is a sequence of state-spec/value pairs as defined by <code>Style.map()</code> , specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.
compound	Specifies how to display the image relative to the text, in the case both text and images options are present. Valid values are : <ul style="list-style-type: none"> — text : display text only — image : display image only — top, bottom, left, right : display image above, below, left of, or right of the text, respectively. — none : the default. display the image if present, otherwise the text.
width	If greater than zero, specifies how much space, in character widths, to allocate for the text label, if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.

Compatibility Options

Option	Description
state	May be set to « normal » or « disabled » to control the « disabled » state bit. This is a write-only option : setting it changes the widget state, but the <code>Widget.state()</code> method does not affect this option.

Widget States

The widget state is a bitmap of independent state flags.

Option	Description
active	The mouse cursor is over the widget and pressing a mouse button will cause some action to occur
disabled	Widget is disabled under program control
focus	Widget has keyboard focus
pressed	Widget is being pressed
selected	« On », « true », or « current » for things like Checkbuttons and radiobuttons
background	Windows and Mac have a notion of an « active » or foreground window. The <i>background</i> state is set for widgets in a background window, and cleared for those in the foreground window
readonly	Widget should not allow user modification
alternate	A widget-specific alternate display format
invalid	The widget's value is invalid

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

ttk.Widget

Besides the methods described below, the `ttk.Widget` supports the methods `tkinter.Widget.cget()` and `tkinter.Widget.configure()`.

class `tkinter.ttk.Widget`

identify (*x*, *y*)

Returns the name of the element at position *x y*, or the empty string if the point does not lie within any element.

x and *y* are pixel coordinates relative to the widget.

instate (*statespec*, *callback=None*, **args*, ***kw*)

Test the widget's state. If a callback is not specified, returns `True` if the widget state matches *statespec* and `False` otherwise. If callback is specified then it is called with *args* if widget state matches *statespec*.

state (*statespec=None*)

Modify or inquire widget state. If *statespec* is specified, sets the widget state according to it and return a new *statespec* indicating which flags were changed. If *statespec* is not specified, returns the currently-enabled state flags.

statespec will usually be a list or a tuple.

25.2.4 Combobox

The `ttk.Combobox` widget combines a text field with a pop-down list of values. This widget is a subclass of `Entry`.

Besides the methods inherited from `Widget` : `Widget.cget()`, `Widget.configure()`, `Widget.identify()`, `Widget.instate()` and `Widget.state()`, and the following inherited from `Entry` : `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`, it has some other methods, described at `ttk.Combobox`.

Options

This widget accepts the following specific options :

Option	Description
<code>exportselection</code>	Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking <code>Misc.selection_get</code> , for example).
<code>justify</code>	Specifies how the text is aligned within the widget. One of « left », « center », or « right ».
<code>height</code>	Specifies the height of the pop-down listbox, in rows.
<code>postcommand</code>	A script (possibly registered with <code>Misc.register</code>) that is called immediately before displaying the values. It may specify which values to display.
<code>state</code>	One of « normal », « readonly », or « disabled ». In the « readonly » state, the value may not be edited directly, and the user can only selection of the values from the dropdown list. In the « normal » state, the text field is directly editable. In the « disabled » state, no interaction is possible.
<code>textvariable</code>	Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See <code>tkinter.StringVar</code> .
<code>valeurs</code>	Specifies the list of values to display in the drop-down listbox.
<code>width</code>	Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget's font.

Virtual events

The combobox widget generates a «**ComboboxSelected**» virtual event when the user selects an element from the list of values.

ttk.Combobox

class tkinter.ttk.Combobox

current (*newindex=None*)

If *newindex* is specified, sets the combobox value to the element position *newindex*. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

get ()

Returns the current value of the combobox.

set (*value*)

Sets the value of the combobox to *value*.

25.2.5 Notebook

Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window.

Options

This widget accepts the following specific options :

Option	Description
height	If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.
padding	Specifies the amount of extra space to add around the outside of the notebook. The padding is a list up to four length specifications left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.
width	If present and greater than zero, specified the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.

Tab Options

There are also specific options for tabs :

Option	Description
state	Either « normal », « disabled » or « hidden ». If « disabled », then the tab is not selectable. If « hidden », then the tab is not shown.
sticky	Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters « n », « s », « e » or « w ». Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the <code>grid()</code> geometry manager.
padding	Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.
text	Specifies a text to be displayed in the tab.
image	Specifies an image to display in the tab. See the option image described in <i>Widget</i> .
compound	Specifies how to display the image relative to the text, in the case both options text and image are present. See <i>Label Options</i> for legal values.
underline	Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if <code>Notebook.enable_traversal()</code> is called.

Tab Identifiers

The `tab_id` present in several methods of `ttk.Notebook` may take any of the following forms :

- An integer between zero and the number of tabs
- The name of a child window
- A positional specification of the form « @x,y », which identifies the tab
- The literal string « current », which identifies the currently-selected tab
- The literal string « end », which returns the number of tabs (only valid for `Notebook.index()`)

Virtual Events

This widget generates a «**NotebookTabChanged**» virtual event after a new tab is selected.

ttk.Notebook

```
class tkinter.ttk.Notebook
```

```
add (child, **kw)
```

Adds a new tab to the notebook.

If window is currently managed by the notebook but hidden, it is restored to its previous position.

See *Tab Options* for the list of available options.

```
forget (tab_id)
```

Removes the tab specified by `tab_id`, unmaps and unmanages the associated window.

```
hide (tab_id)
```

Hides the tab specified by `tab_id`.

The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the `add()` command.

```
identify (x, y)
```

Returns the name of the tab element at position `x, y`, or the empty string if none.

```
index (tab_id)
```

Returns the numeric index of the tab specified by `tab_id`, or the total number of tabs if `tab_id` is the string « end ».

insert (*pos*, *child*, ***kw*)

Inserts a pane at the specified position.

pos is either the string « end », an integer index, or the name of a managed child. If *child* is already managed by the notebook, moves it to the specified position.

See [Tab Options](#) for the list of available options.

select (*tab_id=None*)

Selects the specified *tab_id*.

The associated child window will be displayed, and the previously-selected window (if different) is unmapped.

If *tab_id* is omitted, returns the widget name of the currently selected pane.

tab (*tab_id*, *option=None*, ***kw*)

Query or modify the options of the specific *tab_id*.

If *kw* is not given, returns a dictionary of the tab option values. If *option* is specified, returns the value of that *option*. Otherwise, sets the options to the corresponding values.

tabs ()

Returns a list of windows managed by the notebook.

enable_traversal ()

Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows :

— **Control-Tab** : selects the tab following the currently selected one.

— **Shift-Control-Tab** : selects the tab preceding the currently selected one.

— **Alt-K** : where *K* is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

25.2.6 Progressbar

The `ttk.Progressbar` widget shows the status of a long-running operation. It can operate in two modes : 1) the determinate mode which shows the amount completed relative to the total amount of work to be done and 2) the indeterminate mode which provides an animated display to let the user know that work is progressing.

Options

This widget accepts the following specific options :

Option	Description
orient	One of « horizontal » or « vertical ». Specifies the orientation of the progress bar.
length	Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).
mode	One of « determinate » or « indeterminate ».
maximum	A number specifying the maximum value. Defaults to 100.
valeur	The current value of the progress bar. In « determinate » mode, this represents the amount of work completed. In « indeterminate » mode, it is interpreted as modulo <i>maximum</i> ; that is, the progress bar completes one « cycle » when its value increases by <i>maximum</i> .
variable	A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.
phase	Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

ttk.Progressbar

class `tkinter.ttk.Progressbar`

start (*interval=None*)

Begin autoincrement mode : schedules a recurring timer event that calls `Progressbar.step()` every *interval* milliseconds. If omitted, *interval* defaults to 50 milliseconds.

step (*amount=None*)

Increments the progress bar's value by *amount*.
amount defaults to 1.0 if omitted.

stop ()

Stop autoincrement mode : cancels any recurring timer event initiated by `Progressbar.start()` for this progress bar.

25.2.7 Separator

The `ttk.Separator` widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from `ttk.Widget`.

Options

This widget accepts the following specific option :

Option	Description
<code>orient</code>	One of « horizontal » or « vertical ». Specifies the orientation of the separator.

25.2.8 Sizegrip

The `ttk.Sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from `ttk.Widget`.

Platform-specific notes

- On MacOS X, toplevel windows automatically include a built-in size grip by default. Adding a `Sizegrip` is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel's position was specified relative to the right or bottom of the screen (e.g.), the `Sizegrip` widget will not resize the window.
- This widget supports only « southeast » resizing.

25.2.9 Treeview

The `ttk.Treeview` widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option `columns`. See *Column Identifiers*.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{ }`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in *Scrollable Widget Options* and the methods `Treeview.xview()` and `Treeview.yview()`.

Options

This widget accepts the following specific options :

Option	Description
<code>columns</code>	A list of column identifiers, specifying the number of columns and their names.
<code>displaycolumns</code>	A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string « #all ».
<code>height</code>	Specifies the number of rows which should be visible. Note : the requested width is determined from the sum of the column widths.
<code>padding</code>	Specifies the internal padding for the widget. The padding is a list of up to four length specifications.
<code>selectmode</code>	Controls how the built-in class bindings manage the selection. One of « extended », « browse » or « none ». If set to « extended » (the default), multiple items may be selected. If « browse », only a single item will be selected at a time. If « none », the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.
<code>show</code>	A list containing zero or more of the following values, specifying which elements of the tree to display. — <code>tree</code> : display tree labels in column #0. — <code>headings</code> : display the heading row. The default is « tree headings », i.e., show all elements. Note : Column #0 always refers to the tree column, even if <code>show= »tree »</code> is not specified.

Item Options

The following item options may be specified for items in the insert and item widget commands.

Option	Description
text	The textual label to display for the item.
image	A Tk Image, displayed to the left of the label.
valeurs	The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.
open	True/False value indicating whether the item's children should be displayed or hidden.
tags	A list of tags associated with this item.

Tag Options

The following options may be specified on tags :

Option	Description
foreground	Specifies the text foreground color.
background	Specifies the cell or item background color.
font	Specifies the font to use when drawing text.
image	Specifies the item image, in case the item's image option is empty.

Column Identifiers

Column identifiers take any of the following forms :

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

Notes :

- Item's option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show= »tree » is not specified.

A data column number is an index into an item's option values list ; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, **column #0 always refers to the tree column**.

Virtual Events

The Treeview widget generates the following virtual events.

Event	Description
«TreeviewSelect»	Generated whenever the selection changes.
«TreeviewOpen»	Generated just before settings the focus item to open=True.
«TreeviewClose»	Generated just after setting the focus item to open=False.

The `Treeview.focus()` and `Treeview.selection()` methods can be used to determine the affected item or items.

ttk.Treeview

```
class tkinter.ttk.Treeview
```

bbox (*item*, *column=None*)

Returns the bounding box (relative to the treeview widget's window) of the specified *item* in the form (x, y, width, height).

If *column* is specified, returns the bounding box of that cell. If the *item* is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children (*item=None*)

Returns the list of children belonging to *item*.

If *item* is not specified, returns root children.

set_children (*item*, **newchildren*)

Replaces *item*'s child with *newchildren*.

Children present in *item* that are not present in *newchildren* are detached from the tree. No items in *newchildren* may be an ancestor of *item*. Note that not specifying *newchildren* results in detaching *item*'s children.

column (*column*, *option=None*, ***kw*)

Query or modify the options for the specified *column*.

If *kw* is not given, returns a dict of the column option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are :

- **id** Returns the column name. This is a read-only option.
- **anchor** : **One of the standard Tk anchor values**. Specifies how the text in this column should be aligned with respect to the cell.
- **minwidth** : **width** The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.
- **stretch** : **True/False** Specifies whether the column's width should be adjusted when the widget is resized.
- **width** : **width** The width of the column in pixels.

To configure the tree column, call this with *column* = « #0 »

delete (**items*)

Delete all specified *items* and all their descendants.

The root item may not be deleted.

detach (**items*)

Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

exists (*item*)

Returns **True** if the specified *item* is present in the tree.

focus (*item=None*)

If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or "" if there is none.

heading (*column*, *option=None*, ***kw*)

Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are :

- **text** : **text** The text to display in the column heading.
- **image** : **imageName** Specifies an image to display to the right of the column heading.
- **anchor** : **anchor** Specifies how the heading text should be aligned. One of the standard Tk anchor values.
- **command** : **callback** A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with *column* = « #0 ».

identify (*component*, *x*, *y*)

Returns a description of the specified *component* under the point given by *x* and *y*, or the empty string if no such *component* is present at that position.

identify_row (*y*)

Returns the item ID of the item at position *y*.

identify_column (*x*)

Returns the data column identifier of the cell at position *x*.

The tree column has ID #0.

identify_region (*x*, *y*)

Returns one of :

<i>region</i>	meaning
heading	Tree heading area.
separator	Space between two columns headings.
tree	The tree area.
cell	A data cell.

Availability : Tk 8.6.

identify_element (*x*, *y*)

Returns the element at position *x*, *y*.

Availability : Tk 8.6.

index (*item*)

Returns the integer index of *item* within its parent's list of children.

insert (*parent*, *index*, *iid=None*, ***kw*)

Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. *index* is an integer, or the value « end », specifying where in the list of parent's children to insert the new item. If *index* is less than or equal to zero, the new node is inserted at the beginning; if *index* is greater than or equal to the current number of children, it is inserted at the end. If *iid* is specified, it is used as the item identifier; *iid* must not already exist in the tree. Otherwise, a new unique identifier is generated.

See [Item Options](#) for the list of available points.

item (*item*, *option=None*, ***kw*)

Query or modify the options for the specified *item*.

If no options are given, a dict with options/values for the item is returned. If *option* is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by *kw*.

move (*item*, *parent*, *index*)

Moves *item* to position *index* in *parent*'s list of children.

It is illegal to move an item under one of its descendants. If *index* is less than or equal to zero, *item* is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If *item* was detached it is reattached.

next (*item*)

Returns the identifier of *item*'s next sibling, or "" if *item* is the last child of its parent.

parent (*item*)

Returns the ID of the parent of *item*, or "" if *item* is at the top level of the hierarchy.

prev (*item*)

Returns the identifier of *item*'s previous sibling, or "" if *item* is the first child of its parent.

reattach (*item*, *parent*, *index*)

An alias for [Treeview.move\(\)](#).

see (*item*)

Ensure that *item* is visible.

Sets all of *item*'s ancestors open option to True, and scrolls the widget if necessary so that *item* is within the visible portion of the tree.

selection (*selop=None, items=None*)

If *selop* is not specified, returns selected items. Otherwise, it will act according to the following selection methods.

Deprecated since version 3.6, will be removed in version 3.8 : Using `selection()` for changing the selection state is deprecated. Use the following selection methods instead.

selection_set (**items*)

items becomes the new selection.

Modifié dans la version 3.6 : *items* can be passed as separate arguments, not just as a single tuple.

selection_add (**items*)

Add *items* to the selection.

Modifié dans la version 3.6 : *items* can be passed as separate arguments, not just as a single tuple.

selection_remove (**items*)

Remove *items* from the selection.

Modifié dans la version 3.6 : *items* can be passed as separate arguments, not just as a single tuple.

selection_toggle (**items*)

Toggle the selection state of each item in *items*.

Modifié dans la version 3.6 : *items* can be passed as separate arguments, not just as a single tuple.

set (*item, column=None, value=None*)

With one argument, returns a dictionary of column/value pairs for the specified *item*. With two arguments, returns the current value of the specified *column*. With three arguments, sets the value of given *column* in given *item* to the specified *value*.

tag_bind (*tagname, sequence=None, callback=None*)

Bind a callback for the given event *sequence* to the tag *tagname*. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

tag_configure (*tagname, option=None, **kw*)

Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

tag_has (*tagname, item=None*)

If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

Availability : Tk 8.6

xview (**args*)

Query or modify horizontal position of the treeview.

yview (**args*)

Query or modify vertical position of the treeview.

25.2.10 Ttk Styling

Each widget in `ttk` is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget's class name, but it may be overridden by the widget's style option. If you don't know the class name of a widget, use the method `Misc.winfo_class()` (`somewidget.winfo_class()`).

Voir aussi :

[Tcl'2004 conference presentation](#) This document explains how the theme engine works

class `tkinter.ttk.Style`

This class is used to manipulate the style database.

configure (*style*, *query_opt=None*, ***kw*)

Query or set the default value of the specified option(s) in *style*.

Each key in *kw* is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color :

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
                     background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map (*style*, *query_opt=None*, ***kw*)

Query or sets dynamic values of the specified option(s) in *style*.

Each key in *kw* is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or some other preference. A statespec is a compound of one or more states and then a value.

An example may make it more understandable :

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
         foreground=[('pressed', 'red'), ('active', 'blue')],
         background=[('pressed', '!disabled', 'black'), ('active', 'white')]
        )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

Note that the order of the (states, value) sequences for an option does matter, if the order is changed to `[('active', 'blue'), ('pressed', 'red')]` in the foreground option, for example, the result would be a blue foreground when the widget were in active or pressed states.

lookup (*style*, *option*, *state=None*, *default=None*)

Returns the value specified for *option* in *style*.

If *state* is specified, it is expected to be a sequence of one or more states. If the *default* argument is set, it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default :

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout (*style*, *layoutspec=None*)

Define the widget layout for given *style*. If *layoutspec* is omitted, return the layout specification for given style. *layoutspec*, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in [Layouts](#).

To understand the format, see the following example (it is not intended to do anything useful) :

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
    ("Menubutton.background", None),
    ("Menubutton.button", {"children":
        [ ("Menubutton.focus", {"children":
            [ ("Menubutton.padding", {"children":
                [ ("Menubutton.label", {"side": "left", "expand": 1})]
            })]
        })]
    })],
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create (*elementname*, *etype*, **args*, ***kw*)

Create a new element in the current theme, of the given *etype* which is expected to be either « image », « from » or « vsapi ». The latter is only available in Tk 8.6a for Windows XP and Vista and is not described here.

If « image » is used, *args* should contain the default image name followed by statespec/value pairs (this is the *imagespec*), and *kw* may have the following options :

- **border=padding** padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.
- **height=height** Specifies a minimum height for the element. If less than zero, the base image's height is used as a default.
- **padding=padding** Specifies the element's interior padding. Defaults to border's value if not specified.
- **sticky=spec** Specifies how the image is placed within the final parcel. spec contains zero or more characters « n », « s », « w », or « e ».
- **width=width** Specifies a minimum width for the element. If less than zero, the base image's width is used as a default.

If « from » is used as the value of *etype*, *element_create()* will clone an existing element. *args* is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. *kw* is discarded.

element_names ()

Returns the list of elements defined in the current theme.

element_options (*elementname*)

Returns the list of *elementname*'s options.

theme_create (*themename*, *parent=None*, *settings=None*)

Create a new theme.

It is an error if *themename* already exists. If *parent* is specified, the new theme will inherit styles, elements and layouts from the parent theme. If *settings* are present they are expected to have the same syntax used for *theme_settings()*.

theme_settings (*themename*, *settings*)

Temporarily sets the current theme to *themename*, apply specified *settings* and then restore the previous theme. Each key in *settings* is a style and each value may contain the keys “configure”, “map”, “layout” and “element create” and they are expected to have the same format as specified by the methods *Style.configure()*, *Style.map()*, *Style.layout()* and *Style.element_create()* respectively.

As an example, let's change the Combobox for the default theme a bit :

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
    "TCombobox": {
        "configure": {"padding": 5},
        "map": {
            "background": [("active", "green2"),
                           ("!disabled", "green4")],
            "fieldbackground": [("!disabled", "green3")],
            "foreground": [("focus", "OliveDrab1"),
                           ("!disabled", "OliveDrab2")]
        }
    }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()

Returns a list of all known themes.

theme_use(themename=None)

If *themename* is not given, returns the theme in use. Otherwise, sets the current theme to *themename*, refreshes all widgets and emits a «ThemeChanged» event.

Layouts

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager : given an initial cavity, each element is allocated a parcel. Valid options/values are :

- **side : whichside** Specifies which side of the cavity to place the element ; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.
- **sticky : nswe** Specifies where the element is placed inside its allocated parcel.
- **unit : 0 or 1** If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of `Widget.identify()` et al. It's used for things like scrollbar thumbs with grips.
- **children : [sublayout...]** Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a [Layout](#).

25.3 tkinter.tix — Extension widgets for Tk

Source code : [Lib/tkinter/tix.py](#)

Obsolète depuis la version 3.6 : This Tk extension is unmaintained and should not be used in new code. Use `tkinter.ttk` instead.

The `tkinter.tix` (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The `tkinter.tix` library provides most of the commonly needed widgets that are missing from standard Tk : `HList`, `ComboBox`, `Control` (a.k.a. SpinBox) and an assortment

of scrollable widgets. `tkinter.tix` also includes many more widgets that are generally useful in a wide range of applications : `NoteBook`, `FileEntry`, `PanedWindow`, etc ; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

Voir aussi :

Tix Homepage The home page for Tix. This includes links to additional documentation and downloads.

Tix Man Pages On-line version of the man pages and reference material.

Tix Programming Guide On-line version of the programmer's reference material.

Tix Development Applications Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include **TixInspect**, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.

25.3.1 Using Tix

class `tkinter.tix.Tk` (`screenName=None`, `baseName=None`, `className='Tix'`)

Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.

Classes in the `tkinter.tix` module subclasses the classes in the `tkinter`. The former imports the latter, so to use `tkinter.tix` with Tkinter, all you need to do is to import one module. In general, you can just import `tkinter.tix`, and replace the toplevel call to `tkinter.Tk` with `tix.Tk` :

```
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

To use `tkinter.tix`, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following :

```
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```

If this fails, you have a Tk installation problem which must be resolved before proceeding. Use the environment variable `TIK_LIBRARY` to point to the installed Tix library directory, and make sure you have the dynamic object library (`tix8183.dll` or `libtix8183.so`) in the same directory that contains your Tk dynamic object library (`tk8183.dll` or `libtk8183.so`). The directory with the dynamic object library should also have a file called `pkgIndex.tcl` (case sensitive), which contains the line :

```
package ifneeded Tix 8.1 [list load "[file join $dir tix8183.dll]" Tix]
```

25.3.2 Tix Widgets

Tix introduces over 40 widget classes to the `tkinter` repertoire.

Basic Widgets

class tkinter.tix.Balloon

A **Balloon** that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class tkinter.tix.ButtonBox

The **ButtonBox** widget creates a box of buttons, such as is commonly used for `Ok Cancel`.

class tkinter.tix.ComboBox

The **ComboBox** widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class tkinter.tix.Control

The **Control** widget is also known as the `SpinBox` widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

class tkinter.tix.LabelEntry

The **LabelEntry** widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of « entry-form » type of interface.

class tkinter.tix.LabelFrame

The **LabelFrame** widget packages a frame widget and a label into one mega widget. To create widgets inside a **LabelFrame** widget, one creates the new widgets relative to the `frame` subwidget and manage them inside the `frame` subwidget.

class tkinter.tix.Meter

The **Meter** widget can be used to show the progress of a background job which may take a long time to execute.

class tkinter.tix.OptionMenu

The **OptionMenu** creates a menu button of options.

class tkinter.tix.PopupMenu

The **PopupMenu** widget can be used as a replacement of the `tk_popup` command. The advantage of the **Tix PopupMenu** widget is it requires less application code to manipulate.

class tkinter.tix.Select

The **Select** widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class tkinter.tix.StdButtonBox

The **StdButtonBox** widget is a group of standard buttons for Motif-like dialog boxes.

File Selectors

class tkinter.tix.DirList

The **DirList** widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.DirTree

The **DirTree** widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.DirSelectDialog

The **DirSelectDialog** widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class `tkinter.tix.DirSelectBox`

The `DirSelectBox` is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. `DirSelectBox` stores the directories mostly recently selected into a `ComboBox` widget so that they can be quickly selected again.

class `tkinter.tix.ExFileSelectBox`

The `ExFileSelectBox` widget is usually embedded in a `tixExFileSelectDialog` widget. It provides a convenient method for the user to select files. The style of the `ExFileSelectBox` widget is very similar to the standard file dialog on MS Windows 3.1.

class `tkinter.tix.FileSelectBox`

The `FileSelectBox` is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. `FileSelectBox` stores the files mostly recently selected into a `ComboBox` widget so that they can be quickly selected again.

class `tkinter.tix.FileEntry`

The `FileEntry` widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

Hierarchical ListBox

class `tkinter.tix.HList`

The `HList` widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

class `tkinter.tix.CheckList`

The `CheckList` widget displays a list of items to be selected by the user. `CheckList` acts similarly to the Tk `checkbutton` or `radiobutton` widgets, except it is capable of handling many more items than `checkbuttons` or `radiobuttons`.

class `tkinter.tix.Tree`

The `Tree` widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

Tabular ListBox

class `tkinter.tix.TList`

The `TList` widget can be used to display data in a tabular format. The list entries of a `TList` widget are similar to the entries in the Tk `listbox` widget. The main differences are (1) the `TList` widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

Manager Widgets

class `tkinter.tix.PanedWindow`

The `PanedWindow` widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class `tkinter.tix.ListNoteBook`

The `ListNoteBook` widget is very similar to the `TixNoteBook` widget : it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the `hlist` subwidget.

class `tkinter.tix.NoteBook`

The `NoteBook` widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual « tabs » at the top of the `NoteBook` widget.

Image Types

The `tkinter.tix` module adds :

- `pixmap` capabilities to all `tkinter.tix` and `tkinter` widgets to create color images from XPM files.
- `Compound` image types can be used to create images that consists of multiple horizontal lines ; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a `Tk Button` widget.

Miscellaneous Widgets

class `tkinter.tix.InputOnly`

The `InputOnly` widgets are to accept inputs from the user, which can be done with the `bind` command (Unix only).

Form Geometry Manager

In addition, `tkinter.tix` augments `tkinter` by providing :

class `tkinter.tix.Form`

The `Form` geometry manager based on attachment rules for all `Tk` widgets.

25.3.3 Tix Commands

class `tkinter.tix.tixCommand`

The `tix commands` provide access to miscellaneous elements of `Tix`'s internal state and the `Tix` application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is :

```
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

`tixCommand.tix_configure` (*cnf=None, **kw*)

Query or modify the configuration options of the `Tix` application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s) ; in this case the method returns an empty string. Option may be any of the configuration options.

`tixCommand.tix_cget` (*option*)

Returns the current value of the configuration option given by *option*. Option may be any of the configuration options.

`tixCommand.tix_getbitmap` (*name*)

Locates a bitmap file of the name `name.xpm` or `name` in one of the bitmap directories (see the `tix_addbitmapdir()` method). By using `tix_getbitmap()`, you can avoid hard coding the pathnames

of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character @. The returned value can be used to configure the `bitmap` option of the Tk and Tix widgets.

`tixCommand.tix_addbitmapdir(directory)`

Tix maintains a list of directories under which the `tix_getimage()` and `tix_getbitmap()` methods will search for image files. The standard bitmap directory is `$TIX_LIBRARY/bitmaps`. The `tix_addbitmapdir()` method adds `directory` into this list. By using this method, the image files of an applications can also be located using the `tix_getimage()` or `tix_getbitmap()` method.

`tixCommand.tix_filedialog([dlgclass])`

Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to `tix_filedialog()`. An optional `dlgclass` parameter can be passed as a string to specify what type of file selection dialog widget is desired. Possible options are `tix`, `FileSelectDialog` or `tixExFileSelectDialog`.

`tixCommand.tix_getimage(self, name)`

Locates an image file of the name `name.xpm`, `name.xbm` or `name.ppm` in one of the bitmap directories (see the `tix_addbitmapdir()` method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display : xbm images are chosen on monochrome displays and color images are chosen on color displays. By using `tix_getimage()`, you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the `image` option of the Tk and Tix widgets.

`tixCommand.tix_option_get(name)`

Gets the options maintained by the Tix scheme mechanism.

`tixCommand.tix_resetoptions(newScheme, newFontSet[, newScmPrio])`

Resets the scheme and fontset of the Tix application to `newScheme` and `newFontSet`, respectively. This affects only those widgets created after this call. Therefore, it is best to call the `resetoptions` method before the creation of any widgets in a Tix application.

The optional parameter `newScmPrio` can be given to reset the priority level of the Tk options set by the Tix schemes. Because of the way Tk handles the X option database, after Tix has been imported and initied, it is not possible to reset the color schemes and font sets using the `tix_config()` method. Instead, the `tix_resetoptions()` method must be used.

25.4 tkinter.scrolledtext — Gadget texte avec barre de défilement

Code source : [Lib/tkinter/scrolledtext.py](#)

Le module `tkinter.scrolledtext` fournit une classe, de même nom, implémentant un simple gadget texte avec une barre de défilement verticale, configuré « pour faire ce qu'on attend de lui ». Utiliser `ScrolledText` est beaucoup plus simple que configurer un gadget texte et une barre de défilement. Le constructeur est le même que celui de la classe `tkinter.Text`.

Le gadget texte et la barre de défilement sont regroupés dans une `Frame`, et les méthodes gestionnaires de géométrie `Grid` et `Pack` sont récupérées de l'objet `Frame`. L'objet `ScrolledText` a donc tous les attributs classiques pour la gestion de la géométrie.

Si un contrôle plus fin est nécessaire, les attributs suivants sont disponibles :

`ScrolledText.frame`

Le cadre (objet `Frame`) qui englobe le gadget texte et le gadget de la barre de défilement.

`ScrolledText.vbar`

Le gadget de la barre de défilement.

25.5 IDLE

Code source : [Lib/idlelib/](https://lib.idlelib/)

IDLE est l'environnement de développement et d'apprentissage intégré de Python (*Integrated Development and Learning Environment*).

IDLE a les fonctionnalités suivantes :

- codé à 100% en pur Python, en utilisant l'outil d'interface graphique *tkinter*
- cross-platform : works mostly the same on Windows, Unix, and macOS
- Console Python (interpréteur interactif) avec coloration du code entré, des sorties et des messages d'erreur
- éditeur de texte multi-fenêtres avec annulations multiples, coloration Python, indentation automatique, aide pour les appels de fonction, *autocomplétion*, parmi d'autres fonctionnalités
- recherche dans n'importe quelle fenêtre, remplacement dans une fenêtre d'édition et recherche dans des fichiers multiples (*grep*)
- débogueur avec points d'arrêt persistants, pas-à-pas et visualisation des espaces de nommage locaux et globaux
- configuration, navigateur et d'autres fenêtres de dialogue

25.5.1 Menus

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. On Windows and Linux, each has its own top menu. Each menu documented below indicates which window type it is associated with.

Output windows, such as used for Edit => Find in Files, are a subtype of editor window. They currently have the same top menu but a different default title and context menu.

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidelines.

Menu *File* (Console et Éditeur)

New File Crée une nouvelle fenêtre d'édition.

Open... Ouvre un fichier existant avec une fenêtre de dialogue pour l'ouverture.

Recent Files Ouvre une liste des fichiers récents. Cliquez sur l'un d'eux pour l'ouvrir.

Open Module... Ouvre un module existant (cherche dans *sys.path*).

Class Browser Montre les fonctions, classes et méthodes dans une arborescence pour le fichier en cours d'édition.
Dans la console, ouvre d'abord un module.

Path Browser Affiche les dossiers de *sys.path*, les modules, fonctions, classes et méthodes dans une arborescence.

Save Enregistre la fenêtre active sous le fichier associé, s'il existe. Les fenêtres qui ont été modifiées depuis leur ouverture ou leur dernier enregistrement ont un * avant et après le titre de la fenêtre. S'il n'y a aucun fichier associé, exécute *Save As* à la place.

Save As... Enregistre la fenêtre active avec une fenêtre de dialogue d'enregistrement. Le fichier enregistré devient le nouveau fichier associé pour cette fenêtre.

Save Copy As... Enregistre la fenêtre active sous un fichier différent sans changer le fichier associé.

Print Window Imprime la fenêtre active avec l'imprimante par défaut.

Close Ferme la fenêtre active (demande à enregistrer si besoin).

Exit Ferme toutes les fenêtres et quitte *IDLE* (demande à enregistrer les fenêtres non sauvegardées).

Menu *Edit* (console et éditeur)

Undo Annule le dernier changement dans la fenêtre active. Un maximum de 1000 changements peut être annulé.

Redo Ré-applique le dernier changement annulé dans la fenêtre active.

Cut Copie la sélection dans le presse-papier global ; puis supprime la sélection.

Copy Copie la sélection dans le presse-papier global.

Paste Insère le contenu du presse-papier global dans la fenêtre active.

Les fonctions du presse-papier sont aussi disponibles dans les menus contextuels.

Select All Sélectionne la totalité du contenu de la fenêtre active.

Find... Ouvre une fenêtre de recherche avec de nombreuses options

Find Again Répète la dernière recherche, s'il y en a une.

Find Selection Cherche la chaîne sélectionnée, s'il y en a une.

Find in Files... Ouvre une fenêtre de recherche de fichiers. Présente les résultats dans une nouvelle fenêtre d'affichage.

Replace... Ouvre une fenêtre de recherche et remplacement.

Go to Line Déplace le curseur sur la ligne de numéro demandé et rend cette ligne visible.

Show Completions Open a scrollable list allowing selection of keywords and attributes. See Completions in the Tips sections below.

Expand Word Complète un préfixe que vous avez saisi pour correspondre à un mot complet de la même fenêtre ; recommencez pour obtenir un autre complément.

Show call tip After an unclosed parenthesis for a function, open a small window with function parameter hints.

Show surrounding parens Surligne les parenthèses encadrantes.

Menu *Format* (fenêtre d'édition uniquement)

Indent Region Décale les lignes sélectionnées vers la droite d'un niveau d'indentation (4 espaces par défaut).

Dedent Region Décale les lignes sélectionnées vers la gauche d'un niveau d'indentation (4 espaces par défaut).

Comment Out Region Insère `##` devant les lignes sélectionnées.

Uncomment Region Enlève les `#` ou `##` au début des lignes sélectionnées.

Tabify Region Transforme les blocs d'espaces *au début des lignes* en tabulations. (Note : Nous recommandons d'utiliser des blocs de 4 espaces pour indenter du code Python.)

Untabify Region Transforme *toutes* les tabulations en le bon nombre d'espaces.

Toggle Tabs Ouvre une boîte de dialogue permettant de passer des espaces aux tabulations (et inversement) pour l'indentation.

New Indent Width Ouvre une boîte de dialogue pour changer la taille de l'indentation. La valeur par défaut acceptée par la communauté Python est de 4 espaces.

Format Paragraph Reformate le paragraphe actif, délimité par des lignes vides, en un bloc de commentaires, ou la chaîne de caractères multi-lignes ou ligne sélectionnée en chaîne de caractères. Toutes les lignes du paragraphe seront formatées à moins de N colonnes, avec N valant 72 par défaut.

Strip trailing whitespace Enlève les espaces et autres caractères blancs après le dernier caractère non blanc d'une ligne en appliquant `str.rstrip` à chaque ligne, y compris les lignes avec des chaînes de caractère multi-lignes.

Menu *Run* (fenêtre d'édition uniquement)

Console Python Ouvre ou active la fenêtre de console Python.

Check Module Vérifie la syntaxe du module actuellement ouvert dans la fenêtre d'édition. Si le module n'a pas été enregistré, *IDLE* va soit demander à enregistrer à l'utilisateur, soit enregistrer automatiquement, selon l'option sélectionnée dans l'onglet *General* de la fenêtre de configuration d'*IDLE*. S'il y a une erreur de syntaxe, l'emplacement approximatif est indiqué dans la fenêtre d'édition.

Run Module Do Check Module (above). If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

Menu *Shell* (fenêtre de console uniquement)

View Last Restart Fait défiler la fenêtre de console jusqu'au dernier redémarrage de la console.

Restart Shell Redémarre la console pour nettoyer l'environnement.

Interrupt Execution Arrête un programme en cours d'exécution.

Menu *Debug* (fenêtre de console uniquement)

Go to File/Line Cherche, sur la ligne active et la ligne en-dessous, un nom de fichier et un numéro de ligne. Le cas échéant, ouvre le fichier s'il n'est pas encore ouvert et montre la ligne. Utilisez ceci pour visualiser les lignes de code source référencées dans un *traceback* d'exception et les lignes trouvées par *Find in Files*. Également disponible dans le menu contextuel des fenêtres de console et d'affichage.

Debugger ([dés]activer) Quand cette fonctionnalité est activée, le code saisi dans la console ou exécuté depuis un Éditeur s'exécutera avec le débogueur. Dans l'Éditeur, des points d'arrêt peuvent être placés avec le menu contextuel. Cette fonctionnalité est encore incomplète et plus ou moins expérimentale.

Stack Viewer Montre l'état de la pile au moment de la dernière erreur dans une arborescence, avec accès aux variables locales et globales.

Auto-open Stack Viewer Active ou désactive l'ouverture automatique de l'afficheur de pile après une erreur non gérée.

Menu *Options* (console et éditeur)

Configure IDLE Open a configuration dialog and change preferences for the following : fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions (see below). On macOS, open the configuration dialog by selecting Preferences in the application menu. To use a new built-in color theme (IDLE Dark) with older IDLEs, save it as a new custom theme.

Non-default user settings are saved in a `.idlerc` directory in the user's home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in `.idlerc`.

Code Context (toggle)(Editor Window only) Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window. Clicking a line in this pane exposes that line at the top of the editor.

Menu *Windows* (console et éditeur)

Zoom Height Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog.

The rest of this menu lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Menu *Help* (console et éditeur)

About IDLE Affiche la version, les copyrights, la licence, les crédits, entre autres.

IDLE Help Display this IDLE document, detailing the menu options, basic editing and navigation, and other tips.

Python Docs Accède à la documentation Python locale, si installée, ou ouvre docs.python.org dans un navigateur pour afficher la documentation Python la plus récente.

Turtle Demo Run the `turtledemo` module with example python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab. See the « Help sources » subsection below for more on Help menu choices.

Menus Contextuels

Open a context menu by right-clicking in a window (Control-click on macOS). Context menus have the standard clipboard functions also on the Edit menu.

Cut Copie la sélection dans le presse-papier global ; puis supprime la sélection.

Copy Copie la sélection dans le presse-papier global.

Paste Insère le contenu du presse-papier global dans la fenêtre active.

Les fenêtres d'édition ont aussi des fonctions de points d'arrêt. Les lignes avec un point d'arrêt activé sont marquées. Les points d'arrêt n'ont d'effet que lorsque l'exécution se déroule sous débogueur. Les points d'arrêt pour un fichier sont enregistrés dans le dossier `.idlerc` de l'utilisateur.

Set Breakpoint Place un point d'arrêt sur la ligne active.

Clear Breakpoint Enlève le point d'arrêt sur cette ligne.

Shell and Output windows also have the following.

Go to file/line Même effet que dans le menu *Debug*.

The Shell window also has an output squeezing facility explained in the the *Python Shell window* subsection below.

Squeeze If the cursor is over an output line, squeeze all the output between the code above and the prompt below down to a “Squeezed text” label.

25.5.2 Édition et navigation

Editor windows

IDLE may open editor windows when it starts, depending on settings and how you start IDLE. Thereafter, use the File menu. There can be only one open editor window for a given file.

The title bar contains the name of the file, the full path, and the version of Python and IDLE running the window. The status bar contains the line number (“Ln”) and column number (“Col”). Line numbers start with 1 ; column numbers with 0.

IDLE assumes that files with a known `.py*` extension contain Python code and that other files do not. Run Python code with the Run menu.

Key bindings

In this section, “C” refers to the `Control` key on Windows and Unix and the `Command` key on macOS.

- `Retour arrière` supprime à gauche ; `Suppr` supprime à droite
- `C-Retour arrière` supprime le mot à gauche ; `C-Suppr` supprime le mot à droite
- Utilisez les touches `flèche` et `Page Haut / Page Bas` pour vous déplacer
- `C-Flèche Gauche` et `C-Flèche Droite` déplacent de mot en mot
- `Début/Fin` vont au début / à la fin de la ligne
- `C-Début / C-Fin` vont au début / à la fin du fichier
- Quelques raccourcis *Emacs* utiles sont hérités de *Tcl/Tk* :
 - `C-a` début de ligne
 - `C-e` fin de ligne
 - `C-k` supprime la ligne (mais ne la met pas dans le presse-papier)
 - `C-l` centre la fenêtre autour du point d’insertion
 - `C-b` recule d’un caractère sans le supprimer (habituellement vous pouvez également utiliser les touches flèches pour faire cela)
 - `C-f` avance d’un caractère sans le supprimer (habituellement vous pouvez également utiliser les touches flèches pour faire cela)
 - `C-p` remonte d’une ligne (habituellement vous pouvez également utiliser les touches flèches pour faire cela)
 - `C-d` supprime le caractère suivant

Les raccourcis clavier standards (comme `C-c` pour copier et `C-v` pour coller) peuvent fonctionner. Les raccourcis clavier sont sélectionnés dans la fenêtre de configuration d’*IDLE*.

Indentation automatique

Après une structure d’ouverture de bloc, la prochaine ligne est indentée de 4 espaces (dans la console Python d’une tabulation). Après certains mots-clefs (*break*, *return* etc) la ligne suivante est *déindentée*. Dans une indentation au début de la ligne, `Retour arrière` supprime jusqu’à 4 espaces s’il y en a. `Tab` insère des espaces (dans la console, une tabulation), en nombre dépendant de la configuration. Les tabulations sont actuellement restreintes à quatre espaces à cause de limitations de *Tcl/Tk*.

See also the indent/dedent region commands in the edit menu.

Complétions

Les complétions sont fournies pour les fonctions, classes et attributs de classes incluses par défaut et celles définies par l’utilisateur. Les complétions sont aussi fournies pour les noms de fichiers.

La fenêtre d’auto-complétion (*ACW*, *AutoCompleteWindow*) s’ouvre après un délai prédéfini (deux secondes par défaut) après qu’un « . » ou « ** » (dans une chaîne) un `os.sep` est saisi. Si, après un de ces caractères (éventuellement suivi d’autres caractères), une tabulation est saisie, l’*ACW* s’ouvre immédiatement si une complétion compatible est trouvée.

S’il n’y a qu’une seule complétion possible pour le caractère saisi, un `Tab` fournit cette complétion sans ouvrir l’*ACW*.

« *Show Completions* » force l’ouverture d’une fenêtre de complétion, par défaut `C-espace` ouvre une fenêtre de complétion. Dans une chaîne vide, cette fenêtre contient les fichiers du dossier actif. Sur une ligne vide, elle contient les fonctions et classes intégrées par défaut et définies par l’utilisateur de l’espace de nommage actif, plus tous les modules importés. Si des caractères ont été saisis, l’*ACW* essaie d’être plus spécifique.

Si une chaîne de caractère est saisie, la sélection de l’*ACW* va à l’entrée la plus proche de ces caractères. Saisir un `Tab` saisit la plus longue correspondance non ambiguë dans la console ou l’éditeur. Deux `Tab` à la suite fournissent la sélection de l’*ACW*, de la même manière que la touche « *Entrée* » ou un double-clic. Les touches flèches, `Page Haut/Bas`, la sélection à la souris et la molette de la souris fonctionnent tous sur l’*ACW*.

Les attributs « *cachés* » peuvent être atteints en saisissant le début d'un nom caché après un « `.` », e.g. `»**_`. Ceci permet l'accès aux modules utilisant `__all__` ou aux attributs privés des classes.

Les complétions et la fonctionnalité « **Expand Word* »* peuvent vous faire économiser beaucoup de temps !

Les complétions sont actuellement limitées à ce qui est présent dans les espaces de nommage. Les noms dans une fenêtre d'édition qui ne viennent pas de `__main__` et `sys.modules` ne sont pas trouvés. Exécutez votre module avec vos importations pour corriger cette situation. Notez qu'*IDLE* lui-même place quelques modules dans `sys.modules`, qui peuvent être donc accédés par défaut, comme le module *re*.

Si vous n'aimez pas que l'ACW s'affiche spontanément, vous pouvez simplement augmenter le délai ou désactiver l'extension.

Info-bulles

Une info-bulle est affichée quand vous saisissez (après le nom d'une fonction *accessible*. Une expression de nom peut inclure des points et des tirets bas. L'info-bulle reste affichée jusqu'à ce que vous cliquiez dessus, que le curseur se déplace hors de la zone des arguments, ou que) soit saisi. Quand le curseur est dans la partie « *arguments* » de la définition, le menu ou raccourci affiche une info-bulle.

Une info-bulle contient la signature de la fonction et la première ligne de la *docstring*. Pour les fonctions incluses par défaut sans signature accessible, l'info-bulle contient toutes les lignes jusqu'à la cinquième ligne ou la première ligne vide. Ces détails sont sujets à changement.

L'ensemble des fonctions *accessibles* dépend des modules qui ont été importés dans le processus utilisateur, y compris ceux importés par *IDLE* lui-même et quelles définitions ont été exécutées, le tout depuis le dernier redémarrage.

Par exemple, redémarrez la console et saisissez `itertools.count()`. Une info-bulle s'affiche parce que *IDLE* importe *itertools* dans le processus utilisateur pour son propre usage (ceci pourrait changer). Saisissez `turtle.write()` (et rien ne s'affiche. *IDLE* n'importe pas *turtle*. Le menu ou le raccourci ne font rien non plus. Saisir `import *turtle` puis `turtle.write()` fonctionnera.

Dans l'éditeur, les commandes d'importation n'ont pas d'effet jusqu'à ce que le fichier soit exécuté. Vous pouvez exécuter un fichier après avoir écrit les commandes d'importation au début, ou immédiatement exécuter un fichier existant avant de l'éditer.

Fenêtre de console Python

With IDLE's Shell, one enters, edits, and recalls complete statements. Most consoles and terminals only work with a single physical line at a time.

When one pastes code into Shell, it is not compiled and possibly executed until one hits Return. One may edit pasted code first. If one pastes more than one statement into Shell, the result will be a *SyntaxError* when multiple statements are compiled as if they were one.

The editing features described in previous subsections work when entering code interactively. IDLE's Shell window also responds to the following keys.

- `C-c` interrompt l'exécution de la commande
- `C-d` envoie fin-de-fichier (*EOF*) ; cela ferme la fenêtre s'il est saisi à une invite `>>>`
- `Alt-/` (Compléter le mot) est également utile pour réduire la quantité de texte saisie
- Historique des commandes
 - `Alt-p` retrieves previous command matching what you have typed. On macOS use `C-p`.
 - `Alt-n` retrieves next. On macOS use `C-n`.
 - Entrée sur une des commandes précédentes récupère cette commande

Coloration du texte

IDLE affiche par défaut le texte en noir sur blanc mais colore le texte qui possède une signification spéciale. Pour la console, ceci concerne les sorties de la console et de l'utilisateur ainsi que les erreurs de l'utilisateur. Pour le code Python, dans l'invite de commande de la console ou sur un éditeur, ce sont les mots-clefs, noms de fonctions et de classes incluses par défaut, les noms suivant `class` et `def`, les chaînes de caractères et les commentaires. Pour n'importe quelle fenêtre de texte, ce sont le curseur (si présent), le texte trouvé (s'il y en a) et le texte sélectionné.

La coloration du texte est faite en arrière-plan, donc du texte non coloré est parfois visible. Pour changer les couleurs, utilisez l'onglet *Highlighting* de la fenêtre de configuration d'*IDLE*. Le marquage des points d'arrêt du débogueur dans l'éditeur et du texte dans les dialogues n'est pas configurable.

25.5.3 Démarrage et exécution du code

Quand il est démarré avec l'option `-s`, *IDLE* exécutera le fichier référencé par la variable d'environnement `IDLE*STARTUP` ou `PYTHONSTARTUP`. *IDLE* cherche d'abord `IDLESTARTUP` ; si `IDLESTARTUP` est présent, le fichier référencé est exécuté. Si `IDLESTARTUP` n'est pas présent, alors *IDLE* cherche `PYTHONSTARTUP`. Les fichiers référencés par ces variables d'environnement sont de bons endroits pour stocker des fonctions qui sont utilisées fréquemment depuis la console d'*IDLE* ou pour exécuter des commandes d'importation des modules communs.

De plus, Tk charge lui aussi un fichier de démarrage s'il est présent. Notez que le fichier de Tk est chargé sans condition. Ce fichier additionnel est `.Idle.py` et est recherché dans le dossier personnel de l'utilisateur. Les commandes dans ce fichier sont exécutées dans l'espace de nommage de Tk, donc ce fichier n'est pas utile pour importer des fonctions à utiliser depuis la console Python d'*IDLE*.

Utilisation de la ligne de commande

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-          run stdin in shell (- must be last option before args)
```

S'il y a des arguments :

- Si `-`, `-c` ou `-r` sont utilisés, tous les arguments sont placés dans `sys.argv[1:...]` et `sys.argv[0]` est assigné à `""`, `'-c'`, ou `'-r'`. Aucune fenêtre d'édition n'est ouverte, même si c'est le comportement par défaut fixé dans la fenêtre d'options.
- Sinon, les arguments sont des fichiers ouverts pour édition et `sys.argv` reflète les arguments passés à *IDLE* lui-même.

Échec au démarrage

IDLE utilise un connecteur (*socket* en anglais) pour communiquer entre le processus d'interface graphique d'*IDLE* et le processus d'exécution de code de l'utilisateur. Une connexion doit être établie quand la console démarre ou redémarre (le redémarrage est indiqué par une ligne de division avec « *RESTART* »). Si le processus utilisateur échoue à établir une connexion avec le processus graphique, il affiche une fenêtre d'erreur Tk avec un message « *connexion impossible* » qui redirige l'utilisateur ici. Ensuite, il s'arrête.

Une cause d'échec courant est un fichier écrit par l'utilisateur avec le même nom qu'un module de la bibliothèque standard, comme *random.py* et *tkinter.py*. Quand un fichier de ce genre est enregistré dans le même répertoire qu'un fichier à exécuter, *IDLE* ne peut pas importer le fichier standard. La solution actuelle consiste à renommer le fichier de l'utilisateur.

Même si c'est plus rare qu'avant, un antivirus ou un pare-feu peuvent interrompre la connexion. Si le programme ne peut pas être paramétré pour autoriser la connexion, alors il doit être éteint pour qu'*IDLE* puisse fonctionner. Cette connexion interne est sûre car aucune donnée n'est visible depuis un port extérieur. Un problème similaire est une mauvaise configuration du réseau qui bloque les connexions.

Des problèmes d'installation de Python stoppent parfois *IDLE* : il peut y avoir un conflit de versions ou bien l'installation peut nécessiter des privilèges administrateurs. Si on corrige le conflit, ou qu'on ne peut ou ne veut pas accorder de privilège, il peut être plus facile de désinstaller complètement Python et de recommencer.

Un processus *pythonw.exe* zombie peut être un problème. Sous Windows, utilisez le Gestionnaire des Tâches pour en détecter puis en arrêter un. Parfois, un redémarrage causé par un plantage ou une interruption clavier (Ctrl-C) peut ne pas réussir à se connecter. Fermer la fenêtre d'erreur ou redémarrer la console dans le menu Shell peut résoudre un problème temporaire.

Quand *IDLE* démarre pour la première fois, il essaie de lire la configuration de l'utilisateur dans les fichiers de *~/.idlerc* (~ est le répertoire personnel de l'utilisateur). S'il y a un problème, un message d'erreur devrait être affiché. Mis à part les erreurs aléatoires du disque dur, ceci peut être évité en n'éditant jamais les fichiers directement, mais en utilisant la fenêtre de configuration, dans Options. Quand cela arrive, la solution peut être de supprimer un ou plusieurs des fichiers de configuration.

Si *IDLE* se ferme sans message et qu'il n'a pas été démarré depuis une console, essayez de le démarrer depuis une console (`python -m idlelib`) et regardez si un message apparaît.

Running user code

With rare exceptions, the result of executing Python code with *IDLE* is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.activeCount()` returns 2 instead of 1.

By default, *IDLE* runs user code in a separate OS process rather than in the user interface process that runs the shell and editor. In the execution process, it replaces `sys.stdin`, `sys.stdout`, and `sys.stderr` with objects that get input from and send output to the Shell window. The original values stored in `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__` are not touched, but may be `None`.

When Shell has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. These include system-specific functions that determine whether a key has been pressed and if so, which.

IDLE's standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from `sys.stdin` or `print` or `write` to `sys.stdout` or `sys.stderr`, *IDLE* should be started in a command line window. The secondary subprocess will then be attached to that window for input and output.

If `sys` is reset by user code, such as with `importlib.reload(sys)`, *IDLE*'s changes are lost and input from the keyboard and output to the screen will not work correctly.

User output in Shell

When a program outputs text, the result is determined by the corresponding output device. When IDLE executes user code, `sys.stdout` and `sys.stderr` are connected to the display area of IDLE's Shell. Some of its features are inherited from the underlying Tk Text widget. Others are programmed additions. Where it matters, Shell is designed for development rather than production runs.

For instance, Shell never throws away output. A program that sends unlimited output to Shell will eventually fill memory, resulting in a memory error. In contrast, some system text windows only keep the last *n* lines of output. A Windows console, for instance, keeps a user-settable 1 to 9999 lines, with 300 the default.

Text widgets display a subset of Unicode, the Basic Multilingual Plane (BMP). Which characters get a proper glyph instead of a replacement box depends on the operating system and installed fonts. Newline characters cause following text to appear on a new line, but other control characters are either replaced with a box or deleted. However, `repr()`, which is used for interactive echo of expression values, replaces control characters, some BMP codepoints, and all non-BMP characters with escape codes before they are output.

Normal and error output are generally kept separate (on separate lines) from code input and each other. They each get different highlight colors.

For SyntaxError tracebacks, the normal “^” marking where the error was detected is replaced by coloring the text with an error highlight. When code run from a file causes other exceptions, one may right click on a traceback line to jump to the corresponding line in an IDLE editor. The file will be opened if necessary.

Shell has a special facility for squeezing output lines down to a “Squeezed text” label. This is done automatically for output over *N* lines (*N* = 50 by default). *N* can be changed in the PyShell section of the General page of the Settings dialog. Output with fewer lines can be squeezed by right clicking on the output. This can be useful lines long enough to slow down scrolling.

Squeezed output is expanded in place by double-clicking the label. It can also be sent to the clipboard or a separate view window by right-clicking the label.

Développer des applications *tkinter*

IDLE is intentionally different from standard Python in order to facilitate development of *tkinter* programs. Enter `import tkinter as tk; root = tk.Tk()` in standard Python and nothing appears. Enter the same in IDLE and a tk window appears. In standard Python, one must also enter `root.update()` to see the window. IDLE does the equivalent in the background, about 20 times a second, which is about every 50 milliseconds. Next enter `b = tk.Button(root, text='button');` `b.pack()`. Again, nothing visibly changes in standard Python until one enters `root.update()`.

La plupart des programmes *tkinter* exécutent `root.mainloop()`, qui d'habitude ne renvoie pas jusqu'à ce que l'application *tk* soit détruite. Si le programme est exécuté avec `python -i` ou depuis un éditeur *IDLE*, une invite de commande `>>>` n'apparaît pas tant que `mainloop()` ne termine pas, c'est-à-dire quand il ne reste plus rien avec lequel interagir.

Avec un programme *tkinter* exécuté depuis un éditeur *IDLE*, vous pouvez immédiatement commenter l'appel à `mainloop`. On a alors accès à une invite de commande et on peut interagir en direct avec l'application. Il faut juste se rappeler de réactiver l'appel à `mainloop` lors de l'exécution en Python standard.

Exécution sans sous-processus

Par défaut *IDLE* exécute le code de l'utilisateur dans un sous-processus séparé via un connecteur sur l'interface de la boucle locale. Cette connexion n'est pas visible de l'extérieur et rien n'est envoyé ou reçu d'Internet. Si un pare-feu s'en plaint quand même, vous pouvez l'ignorer.

Si la tentative de connexion par le *socket* échoue, *IDLE* vous le notifie. Ce genre d'échec est parfois temporaire, mais s'il persiste, le problème peut soit venir d'un pare-feu qui bloque la connexion ou d'une mauvaise configuration dans un système particulier. Jusqu'à ce que le problème soit résolu, vous pouvez exécuter *IDLE* avec l'option *-n* de la ligne de commande.

Si *IDLE* est démarré avec l'option *-n* de la ligne de commande, il s'exécute dans un seul processus et ne crée pas de sous-processus pour exécuter le serveur RPC d'exécution de Python. Ceci peut être utile si Python ne peut pas créer de sous-processus ou de connecteur *RPC* sur votre plateforme. Cependant, dans ce mode, le code de l'utilisateur n'est pas isolé de *IDLE* lui-même. De plus, l'environnement n'est pas réinitialisé quand *Run/Run Module (F5)* est sélectionné. Si votre code a été modifié, vous devez *reload*() les modules affectés et ré-importer tous les éléments spécifiques (e.g. *from foo import baz)* pour que les changements prennent effet. Pour toutes ces raisons, il est préférable d'exécuter *IDLE* avec le sous-processus par défaut si c'est possible.

Obsolète depuis la version 3.4.

25.5.4 Aide et préférences

Help sources

Help menu entry « IDLE Help » displays a formatted html version of the IDLE chapter of the Library Reference. The result, in a read-only tkinter text window, is close to what one sees in a web browser. Navigate through the text with a mousewheel, the scrollbar, or up and down arrow keys held down. Or click the TOC (Table of Contents) button and select a section header in the opened box.

Help menu entry « Python Docs » opens the extensive sources of help, including tutorials, available at docs.python.org/x.y, where “x.y” is the currently running Python version. If your system has an off-line copy of the docs (this may be an installation option), that will be opened instead.

Selected URLs can be added or removed from the help menu at any time using the General tab of the Configure IDLE dialog .

Modifier les préférences

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Keys can be user defined; IDLE ships with four built-in key sets. In addition, a user can create a custom key set in the Configure IDLE dialog under the keys tab.

IDLE on macOS

Under System Preferences : Dock, one can set « Prefer tabs when opening documents » to « Always ». This setting is not compatible with the tk/tkinter GUI framework used by IDLE, and it breaks a few IDLE features.

Extensions

IDLE inclut un outil d'extensions. Les préférences pour les extensions peuvent être changées avec l'onglet Extensions de la fenêtre de préférences. Lisez le début de *config-extensions.def* dans le dossier *idlelib* pour plus d'informations. La seule extension actuellement utilisée par défaut est *zzdummy*, un exemple également utilisé pour les tests.

25.6 Autres paquets d'interface graphique utilisateur

Des boîtes à outils multiplateformes (Windows, Mac OS X ou Unix et assimilé) majeures sont disponibles pour Python :

Voir aussi :

PyGObject PyGObject provides introspection bindings for C libraries using **GObject**. One of these libraries is the GTK+ 3 widget set. GTK+ comes with many more widgets than Tkinter provides. An online [Python GTK+ 3 Tutorial](#) is available.

PyGTK *PyGTK* fournit une surcouche pour une version plus ancienne de la bibliothèque, GTK+ 2. Cette dernière fournit une interface orientée objet qui est légèrement plus haut niveau que son équivalent C. Il y a également une surcouche pour **GNOME**. Un [tutoriel](#) en ligne est disponible.

PyQt *PyQt* est une surcouche de la boîte à outils *Qt* basée sur **sip**. *Qt* est un *framework* complet de développement d'interface graphique en C++ , disponible pour Unix, Windows et Mac OS X. **sip** est un outil pour générer une surcouche de classes Python au dessus de bibliothèques C++, et est spécifiquement conçu pour Python.

PySide *PySide* est une nouvelle surcouche de la boîte à outils *Qt*, fournie par Nokia. Comparée à *PyQT*, son système de licence est plus accommodant pour les application non open source.

wxPython *wxPython* est une boîte à outils d'interface graphique multiplateforme pour Python qui est construite autour de la populaire boîte à outils **wxWidgets** (anciennement *wxWindows*). En plus d'un ensemble de composants graphiques complet, *wxPython* fournit des classes pour de la documentation en ligne, de l'aide contextuelle, de l'impression, de la consultation de HTML, du rendu graphique bas niveau, du glisser-déposer, l'accès au système de presse-papier, un DSL de description de ressources en XML et même plus, y compris une collection de modules contribués par la communauté qui grandit sans cesse. Elle fournit un aspect et une expérience native pour les applications sur Windows, Mac OS X et systèmes Unix en utilisant les composants natifs de chaque plateforme quand cela est possible (GTK+ sur les systèmes Unix et assimilés).

PyGTK, *PyQt*, et *wxPython* fournissent tous un aspect et une expérience moderne ainsi que plus de composants graphiques que Tkinter. De plus, il y a de nombreuses autres boîtes à outils pour Python, que ce soit multiplateformes ou spécifiques à une plateforme. Consultez la page sur la [programmation d'interface graphique](#) du Wiki Python pour une liste bien plus complète, ainsi que des liens vers des documents où les différentes boîtes à outils graphiques sont comparées.

Les modules décrits dans ce chapitre vous aident à écrire des logiciels. Par exemple, le module *pydoc* prend un module et génère de la documentation basée sur son contenu. Les modules *doctest* et *unittest* contiennent des cadres applicatifs pour écrire des tests unitaires qui permettent de valider automatiquement le code en vérifiant que chaque résultat attendu est produit. Le programme **2to3** peut traduire du code Python 2.x en Python 3.x.

La liste des modules documentés dans ce chapitre est :

26.1 `typing` — Support for type hints

Nouveau dans la version 3.5.

Code source : [Lib/typing.py](#)

Note : The `typing` module has been included in the standard library on a *provisional basis*. New features might be added and API may change even between minor releases if deemed necessary by the core developers.

This module supports type hints as specified by **PEP 484** and **PEP 526**. The most fundamental support consists of the types *Any*, *Union*, *Tuple*, *Callable*, *TypeVar*, and *Generic*. For full specification please see **PEP 484**. For a simplified introduction to type hints see **PEP 483**.

The function below takes and returns a string and is annotated as follows :

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

In the function `greeting`, the argument `name` is expected to be of type `str` and the return type `str`. Subtypes are accepted as arguments.

26.1.1 Type aliases

A type alias is defined by assigning the type to the alias. In this example, `Vector` and `List[float]` will be treated as interchangeable synonyms :

```
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Type aliases are useful for simplifying complex type signatures. For example :

```
from typing import Dict, Tuple, List

ConnectionOptions = Dict[str, str]
Address = Tuple[str, int]
Server = Tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: List[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
    message: str,
    servers: List[Tuple[Tuple[str, int], Dict[str, str]]] -> None:
    ...
```

Note that `None` as a type hint is a special case and is replaced by `type(None)`.

26.1.2 NewType

Use the `NewType()` helper function to create distinct types :

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

The static type checker will treat the new type as if it were a subclass of the original type. This is useful in helping catch logical errors :

```
def get_user_name(user_id: UserId) -> str:
    ...

# typechecks
user_a = get_user_name(UserId(42351))

# does not typecheck; an int is not a UserId
user_b = get_user_name(-1)
```

You may still perform all `int` operations on a variable of type `UserId`, but the result will always be of type `int`. This lets you pass in a `UserId` wherever an `int` might be expected, but will prevent you from accidentally creating a

UserId in an invalid way :

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

Note that these checks are enforced only by the static type checker. At runtime the statement `Derived = NewType('Derived', Base)` will make `Derived` a function that immediately returns whatever parameter you pass it. That means the expression `Derived(some_value)` does not create a new class or introduce any overhead beyond that of a regular function call.

More precisely, the expression `some_value is Derived(some_value)` is always true at runtime.

This also means that it is not possible to create a subtype of `Derived` since it is an identity function at runtime, not an actual type :

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not typecheck
class AdminUserId(UserId): pass
```

However, it is possible to create a `NewType()` based on a “derived” `NewType` :

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

and typechecking for `ProUserId` will work as expected.

See [PEP 484](#) for more details.

Note : Recall that the use of a type alias declares two types to be *equivalent* to one another. Doing `Alias = Original` will make the static type checker treat `Alias` as being *exactly equivalent* to `Original` in all cases. This is useful when you want to simplify complex type signatures.

In contrast, `NewType` declares one type to be a *subtype* of another. Doing `Derived = NewType('Derived', Original)` will make the static type checker treat `Derived` as a *subclass* of `Original`, which means a value of type `Original` cannot be used in places where a value of type `Derived` is expected. This is useful when you want to prevent logic errors with minimal runtime cost.

Nouveau dans la version 3.5.2.

26.1.3 Callable

Frameworks expecting callback functions of specific signatures might be type hinted using `Callable[[Arg1Type, Arg2Type], ReturnTypes]`.

Par exemple :

```
from typing import Callable

def feeder(get_next_item: Callable[[], str]) -> None:
    # Body
```

(suite sur la page suivante)

(suite de la page précédente)

```
def async_query(on_success: Callable[[int], None],
               on_error: Callable[[int, Exception], None]) -> None:
    # Body
```

It is possible to declare the return type of a callable without specifying the call signature by substituting a literal ellipsis for the list of arguments in the type hint: `Callable[..., ReturnType]`.

26.1.4 Generics

Since type information about objects kept in containers cannot be statically inferred in a generic way, abstract base classes have been extended to support subscription to denote expected types for container elements.

```
from typing import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                  overrides: Mapping[str, str]) -> None: ...
```

Generics can be parameterized by using a new factory available in typing called *TypeVar*.

```
from typing import Sequence, TypeVar

T = TypeVar('T')          # Declare type variable

def first(l: Sequence[T]) -> T:    # Generic function
    return l[0]
```

26.1.5 User-defined generic types

A user-defined class can be defined as a generic class.

```
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[T]):
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

`Generic[T]` as a base class defines that the class `LoggedVar` takes a single type parameter `T`. This also makes `T` valid as a type within the class body.

The *Generic* base class uses a metaclass that defines `__getitem__()` so that `LoggedVar[t]` is valid as a type :

```
from typing import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

A generic type can have any number of type variables, and type variables may be constrained :

```
from typing import TypeVar, Generic
...

T = TypeVar('T')
S = TypeVar('S', int, str)

class StrangePair(Generic[T, S]):
    ...
```

Each type variable argument to *Generic* must be distinct. This is thus invalid :

```
from typing import TypeVar, Generic
...

T = TypeVar('T')

class Pair(Generic[T, T]):    # INVALID
    ...
```

You can use multiple inheritance with *Generic*:

```
from typing import TypeVar, Generic, Sized

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...
```

When inheriting from generic classes, some type variables could be fixed :

```
from typing import TypeVar, Mapping

T = TypeVar('T')

class MyDict(Mapping[str, T]):
    ...
```

In this case `MyDict` has a single parameter, `T`.

Using a generic class without specifying type parameters assumes *Any* for each position. In the following example, `MyIterable` is not generic but implicitly inherits from `Iterable[Any]` :

```
from typing import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
```

User defined generic type aliases are also supported. Examples :

```
from typing import TypeVar, Iterable, Tuple, Union
S = TypeVar('S')
Response = Union[Iterable[S], int]

# Return type here is same as Union[Iterable[str], int]
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[Tuple[T, T]]

def inproduct(v: Vec[T]) -> T: # Same as Iterable[Tuple[T, T]]
    return sum(x*y for x, y in v)
```

The metaclass used by *Generic* is a subclass of *abc.ABCMeta*. A generic class can be an ABC by including abstract methods or properties, and generic classes can also have ABCs as base classes without a metaclass conflict. Generic metaclasses are not supported. The outcome of parameterizing generics is cached, and most types in the typing module are hashable and comparable for equality.

26.1.6 The Any type

A special kind of type is *Any*. A static type checker will treat every type as being compatible with *Any* and *Any* as being compatible with every type.

This means that it is possible to perform any operation or method call on a value of type on *Any* and assign it to any variable :

```
from typing import Any

a = None      # type: Any
a = []        # OK
a = 2         # OK

s = ''        # type: str
s = a         # OK

def foo(item: Any) -> int:
    # Typechecks; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Notice that no typechecking is performed when assigning a value of type *Any* to a more precise type. For example, the static type checker did not report an error when assigning *a* to *s* even though *s* was declared to be of type *str* and receives an *int* value at runtime !

Furthermore, all functions without a return type or parameter types will implicitly default to using *Any* :

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
return data
```

This behavior allows `Any` to be used as an *escape hatch* when you need to mix dynamically and statically typed code.

Contrast the behavior of `Any` with the behavior of `object`. Similar to `Any`, every type is a subtype of `object`. However, unlike `Any`, the reverse is not true : `object` is *not* a subtype of every other type.

That means when the type of a value is `object`, a type checker will reject almost all operations on it, and assigning it to a variable (or using it as a return value) of a more specialized type is a type error. For example :

```
def hash_a(item: object) -> int:
    # Fails; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Typechecks
    item.magic()
    ...

# Typechecks, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Typechecks, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

Use `object` to indicate that a value could be any type in a typesafe manner. Use `Any` to indicate that a value is dynamically typed.

26.1.7 Classes, functions, and decorators

The module defines the following classes, functions and decorators :

class `typing.TypeVar`

Type variable.

Utilisation :

```
T = TypeVar('T') # Can be anything
A = TypeVar('A', str, bytes) # Must be str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function definitions. See class `Generic` for more information on generic types. Generic functions work as follows :

```
def repeat(x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n

def longest(x: A, y: A) -> A:
    """Return the longest of two strings."""
    return x if len(x) >= len(y) else y
```

The latter example's signature is essentially the overloading of `(str, str) -> str` and `(bytes, bytes) -> bytes`. Also note that if the arguments are instances of some subclass of `str`, the return type is still plain `str`.

At runtime, `isinstance(x, T)` will raise `TypeError`. In general, `isinstance()` and `issubclass()` should not be used with types.

Type variables may be marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. See [PEP 484](#) for more details. By default type variables are invariant. Alternatively, a type variable may specify an upper bound using `bound=<type>`. This means that an actual type substituted (explicitly or implicitly) for the type variable must be a subclass of the boundary type, see [PEP 484](#).

`class typing.Generic`

Abstract base class for generic types.

A generic type is typically declared by inheriting from an instantiation of this class with one or more type variables. For example, a generic mapping type might be defined as :

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

This class can then be used as follows :

```
X = TypeVar('X')
Y = TypeVar('Y')

def lookup_name(mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

`class typing.Type(Generic[CT_co])`

A variable annotated with `C` may accept a value of type `C`. In contrast, a variable annotated with `Type[C]` may accept values that are classes themselves – specifically, it will accept the *class object* of `C`. For example :

```
a = 3          # Has type 'int'
b = int        # Has type 'Type[int]'
c = type(a)    # Also has type 'Type[int]'
```

Note that `Type[C]` is covariant :

```
class User: ...
class BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...

# Accepts User, BasicUser, ProUser, TeamUser, ...
def make_new_user(user_class: Type[User]) -> User:
    # ...
    return user_class()
```

The fact that `Type[C]` is covariant implies that all subclasses of `C` should implement the same constructor signature and class method signatures as `C`. The type checker should flag violations of this, but should also allow constructor calls in subclasses that match the constructor calls in the indicated base class. How the type checker is required to handle this particular case may change in future revisions of [PEP 484](#).

The only legal parameters for `Type` are classes, *Any*, *type variables*, and unions of any of these types. For example :


```
def new_non_team_user(user_class: Type[Union[BaseUser, ProUser]]): ...
```

`Type[Any]` is equivalent to `Type` which in turn is equivalent to `type`, which is the root of Python's metaclass hierarchy.

Nouveau dans la version 3.5.2.

```
class typing.Iterable (Generic[T_co])
    A generic version of collections.abc.Iterable.
```

```
class typing.Iterator (Iterable[T_co])
    A generic version of collections.abc.Iterator.
```

```
class typing.Reversible (Iterable[T_co])
    A generic version of collections.abc.Reversible.
```

```
class typing.SupportsInt
    An ABC with one abstract method __int__.
```

```
class typing.SupportsFloat
    An ABC with one abstract method __float__.
```

```
class typing.SupportsComplex
    An ABC with one abstract method __complex__.
```

```
class typing.SupportsBytes
    An ABC with one abstract method __bytes__.
```

```
class typing.SupportsAbs
    An ABC with one abstract method __abs__ that is covariant in its return type.
```

```
class typing.SupportsRound
    An ABC with one abstract method __round__ that is covariant in its return type.
```

```
class typing.Container (Generic[T_co])
    A generic version of collections.abc.Container.
```

```
class typing.Hashable
    An alias to collections.abc.Hashable
```

```
class typing.Sized
    An alias to collections.abc.Sized
```

```
class typing.Collection (Sized, Iterable[T_co], Container[T_co])
    A generic version of collections.abc.Collection
    Nouveau dans la version 3.6.
```

```
class typing.AbstractSet (Sized, Collection[T_co])
    A generic version of collections.abc.Set.
```

```
class typing.MutableSet (AbstractSet[T])
    A generic version of collections.abc.MutableSet.
```

```
class typing.Mapping (Sized, Collection[KT], Generic[VT_co])
    A generic version of collections.abc.Mapping.
```

```
class typing.MutableMapping (Mapping[KT, VT])
    A generic version of collections.abc.MutableMapping.
```

```
class typing.Sequence (Reversible[T_co], Collection[T_co])
    A generic version of collections.abc.Sequence.
```

```
class typing.MutableSequence (Sequence[T])
    A generic version of collections.abc.MutableSequence.
```

class `typing.ByteString` (*Sequence*[*int*])

A generic version of `collections.abc.ByteString`.

This type represents the types `bytes`, `bytearray`, and `memoryview`.

As a shorthand for this type, `bytes` can be used to annotate arguments of any of the types mentioned above.

class `typing.Deque` (*deque*, *MutableSequence*[*T*])

A generic version of `collections.deque`.

Nouveau dans la version 3.6.1.

class `typing.List` (*list*, *MutableSequence*[*T*])

Generic version of `list`. Useful for annotating return types. To annotate arguments it is preferred to use abstract collection types such as `Mapping`, `Sequence`, or `AbstractSet`.

This type may be used as follows :

```
T = TypeVar('T', int, float)

def vec2(x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives(vector: Sequence[T]) -> List[T]:
    return [item for item in vector if item > 0]
```

class `typing.Set` (*set*, *MutableSet*[*T*])

A generic version of `builtins.set`.

class `typing.FrozenSet` (*frozenset*, *AbstractSet*[*T_co*])

A generic version of `builtins.frozenset`.

class `typing.MappingView` (*Sized*, *Iterable*[*T_co*])

A generic version of `collections.abc.MappingView`.

class `typing.KeysView` (*MappingView*[*KT_co*], *AbstractSet*[*KT_co*])

A generic version of `collections.abc.KeysView`.

class `typing.ItemsView` (*MappingView*, *Generic*[*KT_co*, *VT_co*])

A generic version of `collections.abc.ItemsView`.

class `typing.ValuesView` (*MappingView*[*VT_co*])

A generic version of `collections.abc.ValuesView`.

class `typingAwaitable` (*Generic*[*T_co*])

A generic version of `collections.abc.Awaitable`.

class `typing.Coroutine` (*Awaitable*[*V_co*], *Generic*[*T_co* *T_contra*, *V_co*])

A generic version of `collections.abc.Coroutine`. The variance and order of type variables correspond to those of `Generator`, for example :

```
from typing import List, Coroutine
c = None # type: Coroutine[List[str], str, int]
...
x = c.send('hi') # type: List[str]
async def bar() -> None:
    x = await c # type: int
```

class `typing.AsyncIterable` (*Generic*[*T_co*])

A generic version of `collections.abc.AsyncIterable`.

class `typing.AsyncIterator` (*AsyncIterable*[*T_co*])

A generic version of `collections.abc.AsyncIterator`.

class `typing.ContextManager` (*Generic*[*T_co*])
 A generic version of `contextlib.AbstractContextManager`.
 Nouveau dans la version 3.6.

class `typing.AsyncContextManager` (*Generic*[*T_co*])
 An ABC with async abstract `__aenter__()` and `__aexit__()` methods.
 Nouveau dans la version 3.6.

class `typing.Dict` (*dict*, *MutableMapping*[*KT*, *VT*])
 A generic version of `dict`. The usage of this type is as follows :

```
def get_position_in_index(word_list: Dict[str, int], word: str) -> int:
    return word_list[word]
```

class `typing.DefaultDict` (*collections.defaultdict*, *MutableMapping*[*KT*, *VT*])
 A generic version of `collections.defaultdict`.
 Nouveau dans la version 3.5.2.

class `typing.Counter` (*collections.Counter*, *Dict*[*T*, *int*])
 A generic version of `collections.Counter`.
 Nouveau dans la version 3.6.1.

class `typing.ChainMap` (*collections.ChainMap*, *MutableMapping*[*KT*, *VT*])
 A generic version of `collections.ChainMap`.
 Nouveau dans la version 3.6.1.

class `typing.Generator` (*Iterator*[*T_co*], *Generic*[*T_co*, *T_contra*, *V_co*])
 A generator can be annotated by the generic type `Generator[YieldType, SendType, ReturnType]`.
 For example :

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

Note that unlike many other generics in the typing module, the `SendType` of `Generator` behaves contravariantly, not covariantly or invariantly.

If your generator will only yield values, set the `SendType` and `ReturnType` to `None` :

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Alternatively, annotate your generator as having a return type of either `Iterable[YieldType]` or `Iterator[YieldType]` :

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

class `typing.AsyncGenerator` (*AsyncIterator*[*T_co*], *Generic*[*T_co*, *T_contra*])
 An async generator can be annotated by the generic type `AsyncGenerator[YieldType, SendType]`.
 For example :

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

Unlike normal generators, async generators cannot return a value, so there is no `Return` type parameter. As with `Generator`, the `SendType` behaves contravariantly.

If your generator will only yield values, set the `SendType` to `None` :

```
async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

Alternatively, annotate your generator as having a return type of either `AsyncIterable[YieldType]` or `AsyncIterator[YieldType]` :

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

Nouveau dans la version 3.5.4.

class `typing.Text`

`Text` is an alias for `str`. It is provided to supply a forward compatible path for Python 2 code : in Python 2, `Text` is an alias for `unicode`.

Use `Text` to indicate that a value must contain a unicode string in a manner that is compatible with both Python 2 and Python 3 :

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

Nouveau dans la version 3.5.2.

class `typing.IO`

class `typing.TextIO`

class `typing.BinaryIO`

Generic type `IO[AnyStr]` and its subclasses `TextIO(IO[str])` and `BinaryIO(IO[bytes])` represent the types of I/O streams such as returned by `open()`.

class `typing.Pattern`

class `typing.Match`

These type aliases correspond to the return types from `re.compile()` and `re.match()`. These types (and the corresponding functions) are generic in `AnyStr` and can be made specific by writing `Pattern[str]`, `Pattern[bytes]`, `Match[str]`, or `Match[bytes]`.

class `typing.NamedTuple`

Typed version of `namedtuple`.

Utilisation :

```
class Employee(NamedTuple):
    name: str
    id: int
```

C'est équivalent à :

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

To give a field a default value, you can assign to it in the class body :

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

Fields with a default value must come after any fields without a default.

The resulting class has two extra attributes : `_field_types`, giving a dict mapping field names to types, and `_field_defaults`, a dict mapping field names to default values. (The field names are in the `_fields` attribute, which is part of the namedtuple API.)

NamedTuple subclasses can also have docstrings and methods :

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

Backward-compatible usage :

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

Modifié dans la version 3.6 : Added support for [PEP 526](#) variable annotation syntax.

Modifié dans la version 3.6.1 : Added support for default values, methods, and docstrings.

`typing.NewType` (*typ*)

A helper function to indicate a distinct types to a typechecker, see [NewType](#). At runtime it returns a function that returns its argument. Usage :

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

Nouveau dans la version 3.5.2.

`typing.cast` (*typ*, *val*)

Cast a value to a type.

This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don't check anything (we want this to be as fast as possible).

`typing.get_type_hints` (*obj*[, *globals*[, *locals*]])

Return a dictionary containing type hints for a function, method, module or class object.

This is often the same as `obj.__annotations__`. In addition, forward references encoded as string literals are handled by evaluating them in `globals` and `locals` namespaces. If necessary, `Optional[t]` is added for function and method annotations if a default value equal to `None` is set. For a class `C`, return a dictionary constructed by merging all the `__annotations__` along `C.__mro__` in reverse order.

`@typing.overload`

The `@overload` decorator allows describing functions and methods that support multiple different combinations of argument types. A series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method). The `@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition,

while the latter is used at runtime but should be ignored by a type checker. At runtime, calling a `@overload`-decorated function directly will raise `NotImplementedError`. An example of overload that gives a more precise type than can be expressed using a union or a type variable :

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> Tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    <actual implementation>
```

See [PEP 484](#) for details and comparison with other typing semantics.

`@typing.no_type_check`

Decorator to indicate that annotations are not type hints.

This works as class or function *decorator*. With a class, it applies recursively to all methods defined in that class (but not to methods defined in its superclasses or subclasses).

This mutates the function(s) in place.

`@typing.no_type_check_decorator`

Decorator to give another decorator the `no_type_check()` effect.

This wraps the decorator with something that wraps the decorated function in `no_type_check()`.

`typing.Any`

Special type indicating an unconstrained type.

- Every type is compatible with `Any`.
- `Any` is compatible with every type.

`typing.NoReturn`

Special type indicating that a function never returns. For example :

```
from typing import NoReturn

def stop() -> NoReturn:
    raise RuntimeError('no way')
```

Nouveau dans la version 3.6.5.

`typing.Union`

Union type; `Union[X, Y]` means either X or Y.

To define a union, use e.g. `Union[int, str]`. Details :

- The arguments must be types and there must be at least one.
- Unions of unions are flattened, e.g. :

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g. :

```
Union[int] == int # The constructor actually returns int
```

- Redundant arguments are skipped, e.g. :

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g. :

```
Union[int, str] == Union[str, int]
```

- When a class and its subclass are present, the latter is skipped, e.g. :

```
Union[int, object] == object
```

- You cannot subclass or instantiate a union.
- You cannot write `Union[X][Y]`.
- You can use `Optional[X]` as a shorthand for `Union[X, None]`.

typing.Optional

Optional type.

`Optional[X]` is equivalent to `Union[X, None]`.

Note that this is not the same concept as an optional argument, which is one that has a default. An optional argument with a default does not require the `Optional` qualifier on its type annotation just because it is optional. For example :

```
def foo(arg: int = 0) -> None:
    ...
```

On the other hand, if an explicit value of `None` is allowed, the use of `Optional` is appropriate, whether the argument is optional or not. For example :

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

typing.Tuple

Tuple type; `Tuple[X, Y]` is the type of a tuple of two items with the first item of type `X` and the second of type `Y`.

Example: `Tuple[T1, T2]` is a tuple of two elements corresponding to type variables `T1` and `T2`. `Tuple[int, float, str]` is a tuple of an int, a float and a string.

To specify a variable-length tuple of homogeneous type, use literal ellipsis, e.g. `Tuple[int, ...]`. A plain *tuple* is equivalent to `Tuple[Any, ...]`, and in turn to *tuple*.

typing.Callable

Callable type; `Callable[[int], str]` is a function of `(int) -> str`.

The subscription syntax must always be used with exactly two values : the argument list and the return type. The argument list must be a list of types or an ellipsis; the return type must be a single type.

There is no syntax to indicate optional or keyword arguments ; such function types are rarely used as callback types. `Callable[..., ReturnType]` (literal ellipsis) can be used to type hint a callable taking any number of arguments and returning `ReturnType`. A plain *Callable* is equivalent to `Callable[..., Any]`, and in turn to *collections.abc.Callable*.

typing.ClassVar

Special type construct to mark class variables.

As introduced in [PEP 526](#), a variable annotation wrapped in `ClassVar` indicates that a given attribute is intended to be used as a class variable and should not be set on instances of that class. Usage :

```
class Starship:
    stats: ClassVar[Dict[str, int]] = {} # class variable
    damage: int = 10                    # instance variable
```

ClassVar accepts only types and cannot be further subscribed.

ClassVar is not a class itself, and should not be used with *isinstance()* or *issubclass()*. *ClassVar* does not change Python runtime behavior, but it can be used by third-party type checkers. For example, a type checker might flag the following code as an error :

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}    # This is OK
```

Nouveau dans la version 3.5.3.

typing.**AnyStr**

`AnyStr` is a type variable defined as `AnyStr = TypeVar('AnyStr', str, bytes)`.

It is meant to be used for functions that may accept any kind of string without allowing different kinds of strings to mix. For example :

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat(u"foo", u"bar") # Ok, output has type 'unicode'
concat(b"foo", b"bar") # Ok, output has type 'bytes'
concat(u"foo", b"bar") # Error, cannot mix unicode and bytes
```

typing.**TYPE_CHECKING**

A special constant that is assumed to be `True` by 3rd party static type checkers. It is `False` at runtime. Usage :

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

Note that the first type annotation must be enclosed in quotes, making it a « forward reference », to hide the `expensive_mod` reference from the interpreter runtime. Type annotations for local variables are not evaluated, so the second annotation does not need to be enclosed in quotes.

Nouveau dans la version 3.5.2.

26.2 pydoc — Générateur de documentation et système d'aide en ligne

Code source : [Lib/pydoc.py](#)

Le module `pydoc` génère automatiquement de la documentation à partir de modules Python. La documentation peut se présenter sous forme de pages de texte dans la console, rendue dans un navigateur web, ou sauvegardée dans des fichiers HTML.

Pour les modules, classes, fonctions et méthodes, la documentation affichée est tirée de la *docstring* (c.à.d de l'attribut `__doc__`) de l'objet et ce, de manière récursive pour les membres qui peuvent être documentés. S'il n'y a pas de *docstring*, `pydoc` essaie d'obtenir une description à partir du bloc de commentaires juste au-dessus de la définition de la classe, fonction ou méthode du fichier source, ou en haut du module (voir `inspect.getcomments()`).

La fonction native `help()` appelle le système d'aide en ligne dans l'interpréteur Python qui utilise `pydoc` pour générer sa documentation sous forme textuelle dans la console. Cette même documentation peut aussi être consultée à l'extérieur de l'interpréteur Python en lançant `pydoc` dans le terminal du système d'exploitation. Par exemple en lançant

```
pydoc sys
```

dans un terminal, cela affiche la documentation du module `sys` dans un style similaire à la commande Unix `man`. On peut passer comme argument à `pydoc` le nom d'une fonction, d'un module, d'un paquet, ou une référence pointant vers

une classe, une méthode, ou une fonction dans un module ou dans un paquet. Si l'argument passé à **pydoc** est un chemin (c.à.d qu'il contient des séparateurs de chemin tels que la barre oblique / dans Unix), et fait référence à un fichier source Python existant, alors la documentation est générée pour ce fichier.

Note : Afin de trouver des objets et leur documentation, **pydoc** importe le ou les modules à documenter. Par conséquent tout code au niveau du module sera exécuté à cette occasion. Utiliser `if __name__ == '__main__':` évite d'exécuter du code lorsqu'un fichier est appelé directement et non pas importé.

Lorsque l'on affiche une sortie sur la console, **pydoc** essaye de créer une pagination pour faciliter la lecture. Si la variable d'environnement `PAGER` est configurée, **pydoc** utilise sa valeur comme programme de pagination.

Ajouter une option `-w` avant l'argument entraîne l'enregistrement de la documentation HTML générée dans un fichier du répertoire courant au lieu de l'afficher dans la console.

Ajouter une option `-w` avant l'argument cherche les lignes de résumé de tous les modules disponibles pour le mot clé donné comme argument, ceci à la manière de la commande Unix **man**. Les lignes de résumé d'un module sont les premières lignes de sa *docstring*.

Vous pouvez aussi utiliser **pydoc** pour lancer un serveur HTTP sur votre machine locale qui rendra la documentation consultable sur votre navigateur Web. **pydoc -p 1234** lancera un serveur HTTP sur le port 1234, permettant de consulter la documentation à l'adresse `http://localhost:1234/` dans votre navigateur web préféré. En précisant 0 comme numéro de port, un port non utilisé sera aléatoirement alloué.

pydoc -b démarre le serveur et ouvrira en plus un navigateur web vers une page d'index de module. Chaque page affichée a une barre de navigation en haut où vous pouvez *Obtenir* de l'aide sur un élément individuel, *Rechercher* tous les modules avec un mot-clé dans leur ligne de résumé, et aller dans les pages *Index des modules*, *Thèmes* et *Mots clés*.

Quand **pydoc** génère de la documentation, il utilise l'environnement et le chemin courant pour localiser les modules. Ainsi, en invoquant les documents **pydoc spam** en précisant la version du module, vous obtenez le même résultat qu'en lançant l'interpréteur Python et en tapant la commande `import spam`.

La documentation des modules principaux est supposée être hébergée sur `https://docs.python.org/X.Y/library/` et `https://docs.python.org/fr/X.Y/library` pour la version française, où X et Y sont les versions respectivement majeures et mineures de l'interpréteur Python. Ces valeurs peuvent être redéfinies en configurant la variable d'environnement `PYTHONDOS` sur une URL différente ou un répertoire local contenant les pages du manuel de la bibliothèque de référence.

Modifié dans la version 3.2 : Ajout de l'option `-b`.

Modifié dans la version 3.3 : Suppression de l'option `-g`.

Modifié dans la version 3.4 : **pydoc** utilise à présent `inspect.signature()` plutôt que `inspect.getfullargspec()` pour extraire les informations de signatures des *callables*.

26.3 doctest — Test interactive Python examples

Source code : [Lib/doctest.py](#)

The *doctest* module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest :

- To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.

- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of « literate testing » or « executable documentation ».

Here's a complete but small example module :

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
    ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
    ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    26525285981219105863630848000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
    ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n: # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result

if __name__ == "__main__":
    import doctest
```

(suite sur la page suivante)

(suite de la page précédente)

```
doctest.testmod()
```

If you run `example.py` directly from the command line, *doctest* works its magic :

```
$ python example.py
$
```

There's no output ! That's normal, and it means all the examples worked. Pass `-v` to the script, and *doctest* prints a detailed log of what it's trying, and prints a summary at the end :

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

And so on, eventually ending with :

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
      ...
    OverflowError: n too large
ok
2 items passed all tests:
  1 tests in __main__
  8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$
```

That's all you need to know to start making productive use of *doctest* ! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file `Lib/test/test_doctest.py`.

26.3.1 Simple Usage : Checking Examples in Docstrings

The simplest way to start using *doctest* (but not necessarily the way you'll continue to do it) is to end each module `M` with :

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest then examines docstrings in module `M`.

Running the module as a script causes the examples in the docstrings to get executed and verified :

```
python M.py
```

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is `***Test Failed*** N failures.`, where *N* is the number of examples that failed.

Run it with the `-v` switch instead :

```
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing `verbose=True` to `testmod()`, or prohibit it by passing `verbose=False`. In either of those cases, `sys.argv` is not examined by `testmod()` (so passing `-v` or not has no effect).

There is also a command line shortcut for running `testmod()`. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the module name(s) on the command line :

```
python -m doctest -v example.py
```

This will import `example.py` as a standalone module and run `testmod()` on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

For more information on `testmod()`, see section *Basic API*.

26.3.2 Simple Usage : Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the `testfile()` function :

```
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file `example.txt`. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program ! For example, perhaps `example.txt` contains this :

```
The ``example`` module
=====

Using ``factorial``
-----

This is an example text file in reStructuredText format. First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Running `doctest.testfile("example.txt")` then finds the error in this documentation :

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with `testmod()`, `testfile()` won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as `testmod()`.

By default, `testfile()` looks for files in the calling module's directory. See section [Basic API](#) for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like `testmod()`, `testfile()`'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument `verbose`.

There is also a command line shortcut for running `testfile()`. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the file name(s) on the command line :

```
python -m doctest -v example.txt
```

Because the file name does not end with `.py`, `doctest` infers that it must be run with `testfile()`, not `testmod()`.

For more information on `testfile()`, see section [Basic API](#).

26.3.3 How It Works

This section examines in detail how doctest works : which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples ; for information about actually running doctest on these examples, see the following sections.

Which Docstrings Are Examined ?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, if `M.__test__` exists and « is true », it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key `K` in `M.__test__` appears with name

```
<name of M>.__test__.K
```

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

CPython implementation detail : Prior to version 3.4, extension modules written in C were not fully searched by doctest.

How are Docstring Examples Recognized ?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

Any expected output must immediately follow the final '`>>>`' or '`...`' line containing the code, and the expected output (if any) extends to the next '`>>>`' or all-whitespace line.

The fine print :

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.
- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output *are* expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the `NORMALIZE_WHITESPACE` option or *directive* is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom `DocTestParser` class.
- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).
- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them :

```
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

Otherwise, the backslash will be interpreted as part of the string. For example, the `\n` above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string) :

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

- The starting column doesn't matter :

```
>>> assert "Easy!"
>>> import math
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> math.floor(1.9)
1
```

and as many leading whitespace characters are stripped from the expected output as appeared in the initial `'>>>'` line that started the example.

What's the Execution Context ?

By default, each time `doctest` finds a docstring to test, it uses a *shallow copy* of `M`'s globals, so that running tests doesn't change the module's real globals, and so that one test in `M` can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in `M`, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing `globals=your_dict` to `testmod()` or `testfile()` instead.

What About Exceptions ?

No problem, provided that the traceback is the only output produced by the example : just paste in the traceback.¹ Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where `doctest` works hard to be flexible in what it accepts.

Simple example :

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

That `doctest` succeeds if `ValueError` is raised, with the `list.remove(x): x not in list` detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example :

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by `doctest`. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part : the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail :

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

The last three lines (starting with `ValueError`) are compared against the exception's type and detail, and the rest are ignored.

1. Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as :

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
...
ValueError: multi
    line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of `...` is independent of doctest's *ELLIPSIS* option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember :

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects `ValueError: 42 is prime` will pass whether `ValueError` is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.
- Each line of the traceback stack (if present) must be indented further than the first line of the example, *or* start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.
- When the *IGNORE_EXCEPTION_DETAIL* doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.
- The interactive shell omits the traceback header line for some *SyntaxErrors*. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a *SyntaxError* that omits the traceback header, you will need to manually add the traceback header line to your test example.
- For some *SyntaxErrors*, Python displays the character position of the syntax error, using a `^` marker :

```
>>> 1 1
    File "<stdin>", line 1
      1 1
      ^
SyntaxError: invalid syntax
```

Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the `^` marker in the wrong location :

```
>>> 1 1
    File "<stdin>", line 1
      1 1
      ^
SyntaxError: invalid syntax
```

Option Flags

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be bitwise ORed together and passed to various functions. The names can also be used in *doctest directives*, and may be passed to the doctest command line interface via the `-o` option.

Nouveau dans la version 3.4 : The `-o` command line option.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output :

doctest.DONT_ACCEPT_TRUE_FOR_1

By default, if an expected output block contains just 1, an actual output block containing just 1 or just True is considered to be a match, and similarly for 0 versus False. When `DONT_ACCEPT_TRUE_FOR_1` is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting « little integer » output still work in these cases. This option will probably go away, but not for several years.

doctest.DONT_ACCEPT_BLANKLINE

By default, if an expected output block contains a line containing only the string `<BLANKLINE>`, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When `DONT_ACCEPT_BLANKLINE` is specified, this substitution is not allowed.

doctest.NORMALIZE_WHITESPACE

When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. `NORMALIZE_WHITESPACE` is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

doctest.ELLIPSIS

When specified, an ellipsis marker (`. . .`) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it's best to keep usage of this simple. Complicated uses can lead to the same kinds of « oops, it matched too much! » surprises that `. *` is prone to in regular expressions.

doctest.IGNORE_EXCEPTION_DETAIL

When specified, an example that expects an exception passes if an exception of the expected type is raised, even if the exception detail does not match. For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail, e.g., if `TypeError` is raised.

It will also ignore the module name used in Python 3 doctest reports. Hence both of these variations will work with the flag specified, regardless of whether the test is run under Python 2.7 or Python 3.2 (or later versions) :

```
>>> raise CustomError('message')
Traceback (most recent call last):
CustomError: message

>>> raise CustomError('message')
Traceback (most recent call last):
my_module.CustomError: message
```

Note that `ELLIPSIS` can also be used to ignore the details of the exception message, but such a test may still fail based on whether or not the module details are printed as part of the exception name. Using `IGNORE_EXCEPTION_DETAIL` and the details from Python 2.3 is also the only clear way to write a doctest that doesn't care about the exception detail yet continues to pass under Python 2.3 or earlier (those releases do not support *doctest directives* and ignore them as irrelevant comments). For example :

```
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

passes under Python 2.3 and later Python versions with the flag specified, even though the detail changed in Python 2.4 to say « does not » instead of « doesn't ».

Modifié dans la version 3.2 : `IGNORE_EXCEPTION_DETAIL` now also ignores any information relating to the module containing the exception under test.

doctest.SKIP

When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both

documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example's output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily « commenting out » examples.

`doctest.COMPARISON_FLAGS`

A bitmask or'ing together all the comparison flags above.

The second group of options controls how test failures are reported :

`doctest.REPORT_UDIFF`

When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

`doctest.REPORT_CDIFF`

When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

`doctest.REPORT_NDIFF`

When specified, differences are computed by `difflib.Differ`, using the same algorithm as the popular `ndiff.py` utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit 1 where actual output contains letter l, a line is inserted with a caret marking the mismatching column positions.

`doctest.REPORT_ONLY_FIRST_FAILURE`

When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When `REPORT_ONLY_FIRST_FAILURE` is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

`doctest.FAIL_FAST`

When specified, exit after the first failing example and don't attempt to run the remaining examples. Thus, the number of failures reported will be at most 1. This flag may be useful during debugging, since examples after the first failure won't even produce debugging output.

The doctest command line accepts the option `-f` as a shorthand for `-o FAIL_FAST`.

Nouveau dans la version 3.4.

`doctest.REPORTING_FLAGS`

A bitmask or'ing together all the reporting flags above.

There is also a way to register new option flag names, though this isn't useful unless you intend to extend `doctest` internals via subclassing :

`doctest.register_optionflag(name)`

Create a new option flag with a given name, and return the new flag's integer value. `register_optionflag()` can be used when subclassing `OutputChecker` or `DocTestRunner` to create new options that are supported by your subclasses. `register_optionflag()` should always be called using the following idiom :

```
MY_FLAG = register_optionflag('MY_FLAG')
```

Directives

Doctest directives may be used to modify the *option flags* for an individual example. Doctest directives are special Python comments following an example's source code :

```
directive          ::=  "#" "doctest:" directive_options
directive_options  ::=  directive_option ("," directive_option)*
directive_option   ::=  on_or_off directive_option_name
on_or_off          ::=  "+" "|" "-"
directive_option_name ::=  "DONT_ACCEPT_BLANKLINE" "|" "NORMALIZE_WHITESPACE" "|" ...
```

Whitespace is not allowed between the + or - and the directive option name. The directive option name can be any of the option flag names explained above.

An example's doctest directives modify doctest's behavior for that single example. Use + to enable the named behavior, or - to disable it.

For example, this test passes :

```
>>> print(list(range(20)))
[0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so :

```
>>> print(list(range(20)))
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas :

```
>>> print(list(range(20)))
[0,  1,  ...,  18,  19]
```

If multiple directive comments are used for a single example, then they are combined :

```
>>> print(list(range(20)))
...
[0,  1,  ...,  18,  19]
```

As the previous example shows, you can add ... lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line :

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
...
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via + in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via - in a directive can be useful.

Avertissements

`doctest` is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a dict, Python doesn't guarantee that the key-value pairs will be printed in any particular order, so a test like

```
>>> foo()
{"Hermione": "hippogryph", "Harry": "broomstick"}
```

is vulnerable! One workaround is to do

```
>>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}
True
```

instead. Another is to do

```
>>> d = sorted(foo().items())
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C() # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

The *ELLIPSIS* directive gives a nice approach for the last example :

```
>>> C()
<__main__.C instance at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for float formatting, and C libraries vary widely in quality here.

```
>>> 1./7 # risky
0.14285714285714285
>>> print(1./7) # safer
0.142857142857
>>> print(round(1./7, 6)) # much safer
0.142857
```

Numbers of the form $I/2 . ** J$ are safe across all platforms, and I often contrive doctest examples to produce numbers of that form :

```
>>> 3./4 # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

26.3.4 Basic API

The functions `testmod()` and `testfile()` provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections [Simple Usage : Checking Examples in Docstrings](#) and [Simple Usage : Checking Examples in a Text File](#).

```
doctest.testfile(filename, module_relative=True, name=None, package=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, parser=DocTestParser(), encoding=None)
```

All arguments except `filename` are optional, and should be specified in keyword form.

Test examples in the file named `filename`. Return `(failure_count, test_count)`.

Optional argument `module_relative` specifies how the filename should be interpreted :

- If `module_relative` is `True` (the default), then `filename` specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the `package` argument is specified, then it is relative to that package. To ensure OS-independence, `filename` should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If `module_relative` is `False`, then `filename` specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument `name` gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.

Optional argument `package` is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify `package` if `module_relative` is `False`.

Optional argument `globs` gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if `None`, a new empty dict is used.

Optional argument `extraglobs` gives a dict merged into the globals used to execute examples. This works like `dict.update()` : if `globs` and `extraglobs` have a common key, the associated value in `extraglobs` appears in the combined dict. By default, or if `None`, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an `extraglobs` dict mapping the generic name to the subclass to be tested.

Optional argument `verbose` prints lots of stuff if true, and prints only failures if false; by default, or if `None`, it's true if and only if `'-v'` is in `sys.argv`.

Optional argument `report` prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument `optionflags` (default value 0) takes the bitwise OR of option flags. See section [Option Flags](#).

Optional argument `raise_on_error` defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument `parser` specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument `encoding` specifies an encoding that should be used to convert the file to unicode.

```
doctest.testmod(m=None, name=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, exclude_empty=False)
```

All arguments are optional, and all except for `m` should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module `m` (or module `__main__` if `m` is not supplied or is `None`), starting with `m.__doc__`.

Also test examples reachable from dict `m.__test__`, if it exists and is not `None`. `m.__test__` maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module `m` are searched.

Return (failure_count, test_count).

Optional argument *name* gives the name of the module; by default, or if None, `m.__name__` is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using `doctest.master.summarize()` in conjunction with `testmod()` continues to get output for objects with no tests. The *exclude_empty* argument to the newer `DocTestFinder` constructor defaults to true.

Optional arguments *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, and *globs* are the same as for function `testfile()` above, except that *globs* defaults to `m.__dict__`.

`doctest.run_docstring_examples(f, globs, verbose=False, name="NoName", compileflags=None, optionflags=0)`

Test examples associated with object *f*; for example, *f* may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument *globs* is used for the execution context.

Optional argument *name* is used in failure messages, and defaults to "NoName".

If optional argument *verbose* is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument *compileflags* gives the set of flags that should be used by the Python compiler when running the examples. By default, or if None, flags are deduced corresponding to the set of future features found in *globs*.

Optional argument *optionflags* works as for function `testfile()` above.

26.3.5 Unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. `doctest` provides two functions that can be used to create `unittest` test suites from modules and text files containing doctests. To integrate with `unittest` test discovery, include a `load_tests()` function in your test module :

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating `unittest.TestSuite` instances from text files and modules with doctests :

`doctest.DocFileSuite(*paths, module_relative=True, package=None, setUp=None, tearDown=None, globs=None, optionflags=0, parser=DocTestParser(), encoding=None)`

Convert doctest tests from one or more text files to a `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments :

Optional argument *module_relative* specifies how the filenames in *paths* should be interpreted :

- If *module_relative* is `True` (the default), then each filename in *paths* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then each filename in *paths* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in *paths*. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *setUp* specifies a set-up function for the test suite. This is called before running the tests in each file. The *setUp* function will be passed a `DocTest` object. The *setUp* function can access the test globals as the *globals* attribute of the test passed.

Optional argument *tearDown* specifies a tear-down function for the test suite. This is called after running the tests in each file. The *tearDown* function will be passed a `DocTest` object. The *setUp* function can access the test globals as the *globals* attribute of the test passed.

Optional argument *globals* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globals* is a new empty dictionary.

Optional argument *optionflags* specifies the default doctest options for the tests, created by or-ing together individual option flags. See section [Option Flags](#). See function `set_unittest_reportflags()` below for a better way to set reporting options.

Optional argument *parser* specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

The global `__file__` is added to the globals provided to doctests loaded from a text file using `DocFileSuite()`.

```
doctest.DocTestSuite(module=None, globals=None, extraglobals=None, test_finder=None, setUp=None,
                     tearDown=None, checker=None)
```

Convert doctest tests for a module to a `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Optional argument *module* provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument *globals* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globals* is a new empty dictionary.

Optional argument *extraglobals* specifies an extra set of global variables, which is merged into *globals*. By default, no extra globals are used.

Optional argument *test_finder* is the `DocTestFinder` object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function `DocFileSuite()` above.

This function uses the same search technique as `testmod()`.

Modifié dans la version 3.5 : `DocTestSuite()` returns an empty `unittest.TestSuite` if *module* contains no docstrings instead of raising `ValueError`.

Under the covers, `DocTestSuite()` creates a `unittest.TestSuite` out of `doctest.DocTestCase` instances, and `DocTestCase` is a subclass of `unittest.TestCase`. `DocTestCase` isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of `unittest` integration.

Similarly, `DocFileSuite()` creates a `unittest.TestSuite` out of `doctest.DocFileCase` instances, and `DocFileCase` is a subclass of `DocTestCase`.

So both ways of creating a `unittest.TestSuite` run instances of `DocTestCase`. This is important for a subtle reason : when you run `doctest` functions yourself, you can control the `doctest` options in use directly, by passing option flags to `doctest` functions. However, if you're writing a `unittest` framework, `unittest` ultimately controls when and how tests get run. The framework author typically wants to control `doctest` reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through `unittest` to `doctest` test runners.

For this reason, `doctest` also supports a notion of `doctest` reporting flags specific to `unittest` support, via this function :

`doctest.set_unittest_reportflags(flags)`

Set the `doctest` reporting flags to use.

Argument `flags` takes the bitwise OR of option flags. See section [Option Flags](#). Only « reporting flags » can be used. This is a module-global setting, and affects all future doctests run by module `unittest`: the `runTest()` method of `DocTestCase` looks at the option flags specified for the test case when the `DocTestCase` instance was constructed. If no reporting flags were specified (which is the typical and expected case), `doctest`'s `unittest` reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed to the `DocTestRunner` instance created to run the doctest. If any reporting flags were specified when the `DocTestCase` instance was constructed, `doctest`'s `unittest` reporting flags are ignored.

The value of the `unittest` reporting flags in effect before the function was called is returned by the function.

26.3.6 Advanced API

The basic API is a simple wrapper that's intended to make doctest easy to use. It is fairly flexible, and should meet most users' needs; however, if you require more fine-grained control over testing, or wish to extend doctest's capabilities, then you should use the advanced API.

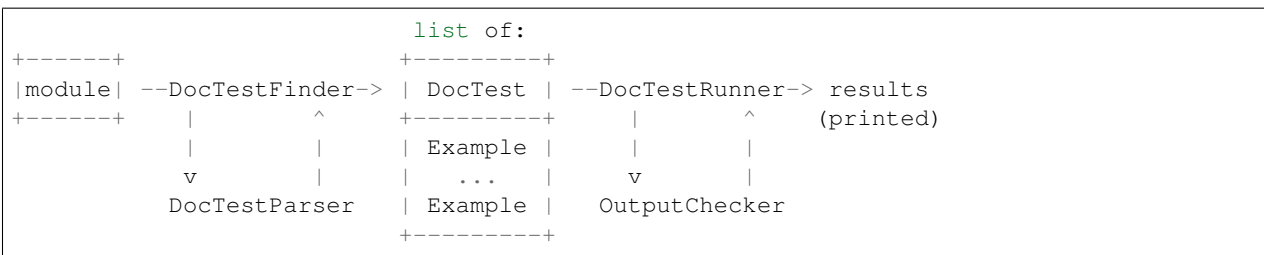
The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- `Example`: A single Python [statement](#), paired with its expected output.
- `DocTest`: A collection of `Examples`, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- `DocTestFinder`: Finds all docstrings in a given module, and uses a `DocTestParser` to create a `DocTest` from every docstring that contains interactive examples.
- `DocTestParser`: Creates a `DocTest` object from a string (such as an object's docstring).
- `DocTestRunner`: Executes the examples in a `DocTest`, and uses an `OutputChecker` to verify their output.
- `OutputChecker`: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:



DocTest Objects

class `doctest.DocTest` (*examples, globs, name, filename, lineno, docstring*)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

`DocTest` defines the following attributes. They are initialized by the constructor, and should not be modified directly.

examples

A list of `Example` objects encoding the individual interactive Python examples that should be run by this test.

globs

The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to

values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in *globs* after the test is run.

name

A string name identifying the *DocTest*. Typically, this is the name of the object or file that the test was extracted from.

filename

The name of the file that this *DocTest* was extracted from; or *None* if the filename is unknown, or if the *DocTest* was not extracted from a file.

lineno

The line number within *filename* where this *DocTest* begins, or *None* if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring

The string that the test was extracted from, or *None* if the string is unavailable, or if the test was not extracted from a string.

Example Objects

class `doctest.Example` (*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)

A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

Example defines the following attributes. They are initialized by the constructor, and should not be modified directly.

source

A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

want

The expected output from running the example's source code (either from stdout, or a traceback in case of exception). *want* ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

exc_msg

The exception message generated by the example, if the example is expected to generate an exception; or *None* if it is not expected to generate an exception. This exception message is compared against the return value of `traceback.format_exception_only()`. *exc_msg* ends with a newline unless it's *None*. The constructor adds a newline if needed.

lineno

The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

indent

The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

options

A dictionary mapping from option flags to *True* or *False*, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the *DocTestRunner*'s *optionflags*). By default, no options are set.

DocTestFinder objects

class `doctest.DocTestFinder` (*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)

A processing class used to extract the *DocTests* that are relevant to a given object, from its docstring and the docstrings of its contained objects. *DocTests* can be extracted from modules, classes, functions, methods, static-methods, classmethods, and properties.

The optional argument *verbose* can be used to display the objects searched by the finder. It defaults to `False` (no output).

The optional argument *parser* specifies the *DocTestParser* object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument *recurse* is false, then *DocTestFinder.find()* will only examine the given object, and not any contained objects.

If the optional argument *exclude_empty* is false, then *DocTestFinder.find()* will include tests for objects with empty docstrings.

DocTestFinder defines the following method :

find (*obj*[, *name*][, *module*][, *globs*][, *extraglobs*])

Return a list of the *DocTests* that are defined by *obj*'s docstring, or by any of its contained objects' docstrings.

The optional argument *name* specifies the object's name; this name will be used to construct names for the returned *DocTests*. If *name* is not specified, then *obj.__name__* is used.

The optional parameter *module* is the module that contains the given object. If the module is not specified or is `None`, then the test finder will attempt to automatically determine the correct module. The object's module is used :

- As a default namespace, if *globs* is not specified.
- To prevent the *DocTestFinder* from extracting *DocTests* from objects that are imported from other modules. (Contained objects with modules other than *module* are ignored.)
- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If *module* is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing *doctest* itself : if *module* is `False`, or is `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each *DocTest* is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each *DocTest*. If *globs* is not specified, then it defaults to the module's *__dict__*, if specified, or `{ }` otherwise. If *extraglobs* is not specified, then it defaults to `{ }`.

DocTestParser objects

class `doctest.DocTestParser`

A processing class used to extract interactive examples from a string, and use them to create a *DocTest* object.

DocTestParser defines the following methods :

get_doctest (*string*, *globs*, *name*, *filename*, *lineno*)

Extract all doctest examples from the given string, and collect them into a *DocTest* object.

globs, *name*, *filename*, and *lineno* are attributes for the new *DocTest* object. See the documentation for *DocTest* for more information.

get_examples (*string*, *name*='<string>')

Extract all doctest examples from the given string, and return them as a list of *Example* objects. Line numbers are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

parse (*string*, *name*='<string>')

Divide the given string into examples and intervening text, and return them as a list of alternating *Examples*

and strings. Line numbers for the *Examples* are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

DocTestRunner objects

class `doctest.DocTestRunner` (*checker=None, verbose=None, optionflags=0*)

A processing class used to execute and verify the interactive examples in a *DocTest*.

The comparison between expected outputs and actual outputs is done by an *OutputChecker*. This comparison may be customized with a number of option flags; see section *Option Flags* for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of *OutputChecker* to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to `TestRunner.run()`; this function will be called with strings that should be displayed. It defaults to `sys.stdout.write`. If capturing the output is not sufficient, then the display output can be also customized by subclassing *DocTestRunner*, and overriding the methods `report_start()`, `report_success()`, `report_unexpected_exception()`, and `report_failure()`.

The optional keyword argument *checker* specifies the *OutputChecker* object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument *verbose* controls the *DocTestRunner*'s verbosity. If *verbose* is `True`, then information is printed about each example, as it is run. If *verbose* is `False`, then only failures are printed. If *verbose* is unspecified, or `None`, then verbose output is used iff the command-line switch `-v` is used.

The optional keyword argument *optionflags* can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section *Option Flags*.

DocTestParser defines the following methods :

report_start (*out, test, example*)

Report that the test runner is about to process the given example. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_success (*out, test, example, got*)

Report that the given example ran successfully. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_failure (*out, test, example, got*)

Report that the given example failed. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_unexpected_exception (*out, test, example, exc_info*)

Report that the given example raised an unexpected exception. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *exc_info* is a tuple containing information about the unexpected exception (as returned by `sys.exc_info()`). *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

run (*test, compileflags=None, out=None, clear_globs=True*)

Run the examples in *test* (a *DocTest* object), and display the results using the writer function *out*.

The examples are run in the namespace `test.globs`. If *clear_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear_globs=False*.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using the *DocTestRunner*'s output checker, and the results are formatted by the *DocTestRunner*.`report_*`() methods.

summarize (*verbose=None*)

Print a summary of all the test cases that have been run by this *DocTestRunner*, and return a *named tuple* *TestResults*(*failed*, *attempted*).

The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the *DocTestRunner*'s verbosity is used.

OutputChecker objects

class `doctest.OutputChecker`

A class used to check the whether the actual output from a doctest example matches the expected output. *OutputChecker* defines two methods : *check_output*(), which compares a given pair of outputs, and returns true if they match; and *output_difference*(), which returns a string describing the differences between two outputs.

OutputChecker defines the following methods :

check_output (*want*, *got*, *optionflags*)

Return True iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section *Option Flags* for more information about option flags.

output_difference (*example*, *got*, *optionflags*)

Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*.

26.3.7 Debugging

Doctest provides several mechanisms for debugging doctest examples :

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, *pdb*.
- The *DebugRunner* class is a subclass of *DocTestRunner* that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
- The *unittest* cases generated by *DocTestSuite*() support the *debug*() method defined by *unittest.TestCase*.
- You can add a call to *pdb.set_trace*() in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose *a.py* contains just this module docstring :

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

Then an interactive Python session may look like this :

```

>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
1      def g(x):
2          print(x+3)
3  ->      import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
1      def f(x):
2  ->      g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>

```

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger :

`doctest.script_from_examples(s)`

Convert text with examples to a script.

Argument *s* is a string containing doctest examples. The string is converted to a Python script, where doctest examples in *s* are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```

import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
    """))

```

displays :

```

# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3

```

This function is used internally by other functions (see below), but can also be useful when you want to transform

an interactive Python session into a Python script.

`doctest.testsource (module, name)`

Convert the doctest for an object to a script.

Argument *module* is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument *name* is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for `script_from_examples()` above. For example, if module `a.py` contains a top-level function `f()`, then

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

prints a script version of function `f()`'s docstring, with doctests converted to code, and the rest placed in comments.

`doctest.debug (module, name, pm=False)`

Debug the doctests for an object.

The *module* and *name* arguments are the same as for function `testsource()` above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, `pdb`.

A shallow copy of `module.__dict__` is used for both local and global execution context.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via `pdb.post_mortem()`, passing the traceback object from the unhandled exception. If *pm* is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate `exec()` call to `pdb.run()`.

`doctest.debug_src (src, pm=False, globs=None)`

Debug the doctests in a string.

This is like function `debug()` above, except that a string containing doctest examples is specified directly, via the *src* argument.

Optional argument *pm* has the same meaning as in function `debug()` above.

Optional argument *globs* gives a dictionary to use as both local and global execution context. If not specified, or None, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

The `DebugRunner` class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially `DebugRunner`'s docstring (which is a doctest !) for more details :

class `doctest.DebugRunner (checker=None, verbose=None, optionflags=0)`

A subclass of `DocTestRunner` that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an `UnexpectedException` exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a `DocTestFailure` exception is raised, containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for `DocTestRunner` in section *Advanced API*.

There are two exceptions that may be raised by `DebugRunner` instances :

exception `doctest.DocTestFailure (test, example, got)`

An exception raised by `DocTestRunner` to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.

`DocTestFailure` defines the following attributes :

`DocTestFailure.test`

The `DocTest` object that was being run when the example failed.

`DocTestFailure.example`

The `Example` that failed.

`DocTestFailure.got`

The example's actual output.

exception `doctest.UnexpectedException` (*test, example, exc_info*)

An exception raised by `DocTestRunner` to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

`UnexpectedException` defines the following attributes :

`UnexpectedException.test`

The `DocTest` object that was being run when the example failed.

`UnexpectedException.example`

The `Example` that failed.

`UnexpectedException.exc_info`

A tuple containing information about the unexpected exception, as returned by `sys.exc_info()`.

26.3.8 Soapbox

As mentioned in the introduction, `doctest` has grown to have three primary uses :

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my `doctest` examples stops working after a « harmless » change.

Doctest also makes an excellent tool for regression testing, especially if you don't skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what's actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It's true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that : the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It's a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests :

- Write text files containing test cases as interactive examples, and test the files using `testfile()` or `DocFileSuite()`. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a `__test__` dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner :


```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                      optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print("{} failures out of {} tests".format(fail, total))
```

Notes

26.4 unittest — Framework de tests unitaires

Code source : `Lib/unittest/__init__.py`

(Si vous êtes déjà familier des concepts de base concernant les tests, vous pouvez souhaiter passer à [la liste des méthodes](#).)

Le cadre applicatif de tests unitaires `unittest` était au départ inspiré par *JUnit* et ressemble aux principaux *frameworks* de tests unitaires des autres langages. Il gère l'automatisation des tests, le partage de code pour la mise en place et la finalisation des tests, l'agrégation de tests en collections, et l'indépendance des tests par rapport au *framework* utilisé.

Pour y parvenir, `unittest` gère quelques concepts importants avec une approche orientée objet :

aménagement de test (*fixture*) Un *aménagement de test* (**fixture**) désigne la préparation nécessaire au déroulement d'un ou plusieurs tests, et toutes les actions de nettoyage associées. Cela peut concerner, par exemple, la création de bases de données temporaires ou mandataires, de répertoires, ou le démarrage d'un processus serveur.

scénario de test Un *scénario de test* est l'élément de base des tests. Il attend une réponse spécifique pour un ensemble particulier d'entrées. `unittest` fournit une classe de base, `TestCase`, qui peut être utilisée pour créer de nouveaux scénarios de test.

suite de tests Une *suite de tests* est une collection de scénarios de test, de suites de tests ou les deux. Cela sert à regrouper les tests qui devraient être exécutés ensemble.

lanceur de tests Un *lanceur de tests* est un composant qui orchestre l'exécution des tests et fournit le résultat pour l'utilisateur. Le lanceur peut utiliser une interface graphique, une interface textuelle, ou renvoie une valeur spéciale pour indiquer les résultats de l'exécution des tests.

Voir aussi :

Module `doctest` Un autre module de test adoptant une approche très différente.

Simple Smalltalk Testing : With Patterns Le papier original de Kent Beck sur les *frameworks* de test utilisant le modèle sur lequel s'appuie `unittest`.

Nose and pytest Des *frameworks* tierces de tests unitaires avec une syntaxe allégée pour l'écriture des tests. Par exemple, `assert func(10) == 42`.

The Python Testing Tools Taxonomy Une liste étendue des outils de test pour Python comprenant des *frameworks* de tests fonctionnels et des bibliothèques d'objets simulés (*mocks*).

Testing in Python Mailing List Un groupe de discussion dédié aux tests, et outils de test, en Python.

Le script `Tools/unittestgui/unittestgui.py` dans la distribution source de Python est un outil avec une interface graphique pour découvrir et exécuter des tests. Il est principalement conçu pour être facile d'emploi pour les débutants en matière de tests unitaires. Pour les environnements de production il est recommandé que les tests soient pilotés par un système d'intégration continue comme [Buildbot](#), [Jenkins](#) ou [Hudson](#).

26.4.1 Exemple basique

Le module `unittest` fournit un riche ensemble d'outils pour construire et lancer des tests. Cette section montre qu'une petite partie des outils suffit pour satisfaire les besoins de la plupart des utilisateurs.

Voici un court script pour tester trois méthodes de `string` :

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

Un scénario de test est créé comme classe-fille de `unittest.TestCase`. Les trois tests individuels sont définis par des méthodes dont les noms commencent par les lettres `test`. Cette convention de nommage signale au lanceur de tests quelles méthodes sont des tests.

Le cœur de chaque test est un appel à `assertEqual()` pour vérifier un résultat attendu; `assertTrue()` ou `assertFalse()` pour vérifier une condition; ou `assertRaises()` pour vérifier qu'une exception particulière est levée. Ces méthodes sont utilisées à la place du mot-clé `assert` pour que le lanceur de tests puisse récupérer les résultats de tous les tests et produire un rapport.

Les méthodes `setUp()` et `tearDown()` vous autorisent à définir des instructions qui seront exécutées avant et après chaque méthode test. Elles sont davantage détaillées dans la section [Organiser le code de test](#).

Le bloc final montre une manière simple de lancer les tests. `unittest.main()` fournit une interface en ligne de commande pour le script de test. Lorsqu'il est lancé en ligne de commande, le script ci-dessus produit une sortie qui ressemble à ceci :

```
...
-----
Ran 3 tests in 0.000s

OK
```

Passer l'option `-v` à votre script de test informera `unittest.main()` qu'il doit fournir un niveau plus important de détails, et produit la sortie suivante :

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok
```

```
-----
Ran 3 tests in 0.001s
```

```
OK
```

Les exemples ci-dessus montrent les fonctionnalités d’*unittest* les plus communément utilisées et qui sont suffisantes pour couvrir les besoins courants en matière de test. Le reste de la documentation explore l’ensemble complet des fonctionnalités depuis les premiers principes.

26.4.2 Interface en ligne de commande

Le module *unittest* est utilisable depuis la ligne de commande pour exécuter des tests à partir de modules, de classes ou même de méthodes de test individuelles :

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

La commande accepte en argument une liste de n’importe quelle combinaison de noms de modules et de noms de classes ou de méthodes entièrement qualifiés.

Les modules de test peuvent également être spécifiés par un chemin de fichier :

```
python -m unittest tests/test_something.py
```

Cette fonctionnalité permet d’utiliser la complétion de l’interpréteur de commandes système (*le shell*) pour spécifier le module de test. Le chemin est converti en nom de module en supprimant le `.py` et en convertissant les séparateurs de chemin en `“.”`. Si vous voulez exécuter un fichier test qui n’est pas importable en tant que module, exécutez directement le fichier.

Pour obtenir plus de détails lors de l’exécution utilisez l’option `-v` (plus de verbosité) :

```
python -m unittest -v test_module
```

Quand la commande est exécutée sans arguments *Découverte des tests* est lancée :

```
python -m unittest
```

Pour afficher la liste de toutes les options de la commande utilisez l’option `-h` :

```
python -m unittest -h
```

Modifié dans la version 3.2 : Dans les versions antérieures, il était seulement possible d’exécuter des méthodes de test individuelles et non des modules ou des classes.

Options de la ligne de commande

Le programme : `unittest` gère ces options de la ligne de commande :

-b, --buffer

Les flux de sortie et d'erreur standards sont mis en mémoire tampon pendant l'exécution des tests. L'affichage produit par un test réussi n'est pas pris en compte. Les sorties d'affichages d'un test en échec ou en erreur sont conservés et ajoutés aux messages d'erreur.

-c, --catch

Utiliser `Control-C` pendant l'exécution des tests attend que le test en cours se termine, puis affiche tous les résultats obtenus jusqu'ici. Une seconde utilisation de `Control-C` provoque l'exception normale `KeyboardInterrupt`.

Voir *Signal Handling* pour les fonctions qui utilisent cette fonctionnalité.

-f, --failfast

Arrête l'exécution des tests lors du premier cas d'erreur ou d'échec.

--locals

Affiche les variables locales dans les traces d'appels.

Nouveau dans la version 3.2 : Les options de ligne de commande `-b`, `-c` et `-f` ont été ajoutées.

Nouveau dans la version 3.5 : Ajout de l'option de ligne de commande `--locals`.

La ligne de commande peut également être utilisée pour découvrir les tests, pour exécuter tous les tests dans un projet ou juste un sous-ensemble.

26.4.3 Découverte des tests

Nouveau dans la version 3.2.

`Unittest` prend en charge une découverte simple des tests. Afin d'être compatible avec le système de découverte de tests, tous les fichiers de test doivent être des modules ou des paquets (incluant des *paquets-espaces de nommage*) importables du répertoire du projet (cela signifie que leurs noms doivent être des identifiants valables).

La découverte de tests est implémentée dans `TestLoader.discover()`, mais peut également être utilisée depuis la ligne de commande. Par exemple :

```
cd project_directory
python -m unittest discover
```

Note : Comme raccourci, `python -m unittest` est l'équivalent de `python -m unittest discover`. Pour passer des arguments au système de découverte des tests, la sous-commande `discover` doit être utilisée explicitement.

La sous-commande `discover` a les options suivantes :

-v, --verbose

Affichage plus détaillé

-s, --start-directory directory

Répertoire racine pour démarrer la découverte (. par défaut).

-p, --pattern pattern

Motif de détection des fichiers de test (`test*.py` par défaut)

-t, --top-level-directory directory

Dossier du premier niveau du projet (Par défaut le dossier de départ)

Les options `-s`, `-p` et `-t` peuvent être passées en arguments positionnels dans cet ordre. Les deux lignes de commande suivantes sont équivalentes :

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

Il est aussi possible de passer un nom de paquet plutôt qu'un chemin, par exemple `monprojet.souspaquet.test`, comme répertoire racine. Le nom du paquet fourni est alors importé et son emplacement sur le système de fichiers est utilisé comme répertoire racine.

Prudence : Le mécanisme de découverte charge les tests en les important. Une fois que le système a trouvé tous les fichiers de tests du répertoire de démarrage spécifié, il transforme les chemins en noms de paquets à importer. Par exemple `truc/bidule/machin.py` est importé sous `truc.bidule.machin`.

Si un paquet est installé globalement et que le mécanisme de découverte de tests est effectué sur une copie différente du paquet, l'importation *peut* se produire à partir du mauvais endroit. Si cela arrive, le système émet un avertissement et se termine.

Si vous donnez le répertoire racine sous la forme d'un nom de paquet plutôt que d'un chemin d'accès à un répertoire, alors Python suppose que l'emplacement à partir duquel il importe est l'emplacement que vous voulez, vous ne verrez donc pas l'avertissement.

Les modules de test et les paquets peuvent adapter le chargement et la découverte des tests en utilisant le protocole *load_tests protocol*.

Modifié dans la version 3.4 : La découverte de tests prend en charge *les paquets-espaces de nommage*.

26.4.4 Organiser le code de test

Les éléments de base des tests unitaires sont les *scénarios de tests* (*test cases* en anglais) — Des scénarios uniques qui sont mis en place et exécutés pour vérifier qu'ils sont corrects. Dans `unittest`, les scénarios de test sont représentés par des instances de `unittest.TestCase`. Pour créer vos propres scénarios de test, vous devez écrire des sous-classes de `TestCase` ou utiliser `FunctionTestCase`.

Le code de test d'une instance de `TestCase` doit être entièrement autonome, de sorte qu'il puisse être exécuté soit de manière isolée, soit en combinaison arbitraire avec un nombre quelconque d'autres scénarios de test.

La sous-classe `TestCase` la plus simple va tout simplement implémenter une méthode de test (c'est-à-dire une méthode dont le nom commence par `test`) afin d'exécuter un code de test spécifique :

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Notez que pour tester quelque chose, on utilise l'une des méthodes `assert*()` fournies par la classe de base `TestCase`. Si le test échoue, une exception est levée avec un message explicatif, et `unittest` identifie ce scénario de test comme un *échec*. Toute autre exception est traitée comme une *erreur*.

Les tests peuvent être nombreux et leur mise en place peut être répétitive. Heureusement, on peut factoriser le code de mise en place en implémentant une méthode appelée `setUp()`, que le système de test appelle automatiquement pour chaque test exécuté :

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                          'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                          'wrong size after resize')
```

Note : L'ordre dans lequel les différents tests sont exécutés est déterminé en classant les noms des méthodes de test en fonction de la relation d'ordre des chaînes de caractères .

Si la méthode `setUp()` lève une exception pendant l'exécution du test, le système considère que le test a subi une erreur, et la méthode test n'est pas exécutée.

De même, on peut fournir une méthode `tearDown()` qui nettoie après l'exécution de la méthode de test :

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

Si `setUp()` a réussi, `tearDown()` est exécutée, que la méthode de test ait réussi ou non.

Such a working environment for the testing code is called a *test fixture*. A new `TestCase` instance is created as a unique test fixture used to execute each individual test method. Thus `setUp()`, `tearDown()`, and `__init__()` will be called once per test.

Il est recommandé d'utiliser `TestCase` pour regrouper les tests en fonction des fonctionnalités qu'ils testent. `unittest` fournit un mécanisme pour cela : la *suite de tests*, représentée par `TestSuite` du module `unittest`. Dans la plupart des cas, appeler `unittest.main()` fait correctement les choses et trouve tous les scénarios de test du module pour vous et les exécute.

Cependant, si vous voulez personnaliser la construction de votre suite de tests, vous pouvez le faire vous-même :

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

Vous pouvez placer les définitions des scénarios de test et des suites de test dans le même module que le code à tester (tel que `composant.py`), mais il y a plusieurs avantages à placer le code de test dans un module séparé, tel que

`test_composant.py` :

- Le module de test peut être exécuté indépendamment depuis la ligne de commande.
- Le code de test est plus facilement séparable du code livré.
- La tentation est moins grande de changer le code de test pour l'adapter au code qu'il teste sans avoir une bonne raison.
- Le code de test doit être modifié beaucoup moins souvent que le code qu'il teste.
- Le code testé peut être réusiné plus facilement.
- Les tests pour les modules écrits en C doivent de toute façon être dans des modules séparés, alors pourquoi ne pas être cohérent ?
- Si la stratégie de test change, il n'est pas nécessaire de changer le code source.

26.4.5 Réutilisation d'ancien code de test

Certains utilisateurs constatent qu'ils ont du code de test existant qu'ils souhaitent exécuter à partir de `unittest`, sans convertir chaque ancienne fonction de test en une sous-classe de `TestCase`.

Pour cette raison, `unittest` fournit une classe `FunctionTestCase`. Cette sous-classe de `TestCase` peut être utilisée pour encapsuler une fonction de test existante. Des fonctions de mise en place (*setUp*) et de démantèlement (*tearDown*) peuvent également être fournies.

Étant donnée la fonction de test suivante :

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

on peut créer une instance de scénario de test équivalente, avec des méthodes optionnelles de mise en place et de démantèlement :

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

Note : Même si la classe `FunctionTestCase` peut être utilisée pour convertir rapidement une base de test existante vers un système basé sur `unittest`, cette approche n'est pas recommandée. Prendre le temps de bien configurer les sous-classes de `TestCase` simplifiera considérablement les futurs réusinages des tests.

Dans certains cas, les tests déjà existants ont pu être écrits avec le module `doctest`. Dans ce cas, `doctest` fournit une classe `DocTestSuite` qui peut construire automatiquement des instances de la classe `unittest.TestSuite` depuis des tests basés sur le module `doctest`.

26.4.6 Ignorer des tests et des erreurs prévisibles

Nouveau dans la version 3.1.

`Unittest` permet d'ignorer des méthodes de test individuelles et même des classes entières de tests. De plus, il prend en charge le marquage d'un test comme étant une « erreur prévue ». Un test qui est cassé et qui échoue, mais qui ne doit pas être considéré comme un échec dans la classe `TestResult`.

Ignorer un test consiste à utiliser le *décorateur* `skip()` ou une de ses variantes conditionnelles.

Un exemple de tests à ignorer :

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

Ceci est le résultat de l'exécution de l'exemple ci-dessus en mode verbeux :

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

-----
Ran 3 tests in 0.005s

OK (skipped=3)
```

Les classes peuvent être ignorées tout comme les méthodes :

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

La méthode `TestCase.setUp()` permet également d'ignorer le test. Ceci est utile lorsqu'une ressource qui doit être configurée n'est pas disponible.

Les erreurs prévisibles utilisent le décorateur `expectedFailure()`

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

Il est facile de faire ses propres décorateurs en créant un décorateur qui appelle `skip()` sur le test que vous voulez ignorer. Par exemple, ce décorateur ignore le test à moins que l'objet passé ne possède un certain attribut :

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

Les décorateurs suivants implémentent le système d'omission des tests et les erreurs prévisibles :

`@unittest.skip(reason)`

Ignore sans condition le test décoré. *La raison* doit décrire la raison pour laquelle le test est omis.

`@unittest.skipIf (condition, reason)`
Ignore le test décoré si la *condition* est vraie.

`@unittest.skipUnless (condition, reason)`
Ignore le test décoré sauf si la *condition* est vraie.

`@unittest.expectedFailure`
Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.

exception `unittest.SkipTest (reason)`
Cette exception est levée pour ignorer un test.
Habituellement, on utilise `TestCase.skipTest()` ou l'un des décorateurs d'omission au lieu de le lever une exception directement.

Les tests ignorés ne lancent ni `setUp()` ni `tearDown()`. Les classes ignorées ne lancent ni `setUpClass()` ni `tearDownClass()`. Les modules sautés n'ont pas `setUpModule()` ou `tearDownModule()` d'exécutés.

26.4.7 Distinguer les itérations de test à l'aide de sous-tests

Nouveau dans la version 3.4.

When there are very small differences among your tests, for instance some parameters, unittest allows you to distinguish them inside the body of a test method using the `subTest()` context manager.

Par exemple, le test suivant :

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

produit le résultat suivant :

```
=====
FAIL: test_even (__main__.NumbersTest) (i=1)
-----
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

=====
FAIL: test_even (__main__.NumbersTest) (i=3)
-----
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

=====
FAIL: test_even (__main__.NumbersTest) (i=5)
-----
Traceback (most recent call last):
```

(suite sur la page suivante)

(suite de la page précédente)

```
File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

Sans l'utilisation d'un sous-test, l'exécution se termine après le premier échec, et l'erreur est moins facile à diagnostiquer car la valeur de `i` ne s'affiche pas :

```
=====
FAIL: test_even (__main__.NumbersTest)
-----
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.4.8 Classes et fonctions

Cette section décrit en détail l'API de `unittest`.

Scénarios de tests

class `unittest.TestCase` (*methodName='runTest'*)

Les instances de la classe `TestCase` représentent des tests logiques unitaires dans l'univers `unittest`. Cette classe est conçue pour être utilisée comme classe de base. Les scénarios de tests sont à implémenter en héritant de cette classe. La classe implémente l'interface nécessaire au lanceur de tests pour lui permettre de les exécuter ainsi que les méthodes que le code de test peut utiliser pour vérifier et signaler les différents types d'erreurs.

Chaque instance de la classe `TestCase` n'exécute qu'une seule méthode de base : la méthode nommée *methodName*. Dans la plupart des utilisations de la classe `TestCase`, vous n'avez pas à changer le nom de la méthode, ni à réimplémenter la méthode `runTest()`.

Modifié dans la version 3.2 : La classe `TestCase` peut désormais être utilisée sans passer de paramètre *methodName*. Cela facilite l'usage de `TestCase` dans l'interpréteur interactif.

Les instances de la classe `TestCase` fournissent trois groupes de méthodes : un groupe utilisé pour exécuter le test, un autre utilisé par l'implémentation du test pour vérifier les conditions et signaler les échecs, et quelques méthodes de recherche permettant de recueillir des informations sur le test lui-même.

Les méthodes du premier groupe (exécution du test) sont :

setUp()

Méthode appelée pour réaliser la mise en place du test. Elle est exécutée immédiatement avant l'appel de la méthode de test ; à l'exception de `AssertionError` ou `SkipTest`, toute exception levée par cette méthode est considérée comme une erreur et non pas comme un échec du test. L'implémentation par défaut ne fait rien.

tearDown()

Méthode appelée immédiatement après l'appel de la méthode de test et l'enregistrement du résultat. Elle est appelée même si la méthode de test a levé une exception. De fait, l'implémentation d'un sous-classes doit être fait avec précaution si vous vérifiez l'état interne de la classe. Toute exception, autre que `AssertionError` ou `SkipTest`, levée par cette méthode est considérée comme une erreur supplémentaire plutôt que comme un échec du test (augmentant ainsi le nombre total des erreurs signalées). Cette méthode est appelée uniquement si l'exécution de `setUp()` est réussie quel que soit le résultat de la méthode de test. L'implémentation par défaut ne fait rien.

setUpClass()

A class method called before tests in an individual class are run. `setUpClass` is called with the class as the only argument and must be decorated as a `classmethod()` :

```
@classmethod
def setUpClass(cls):
    ...
```

Voir *Class and Module Fixtures* pour plus de détails.

Nouveau dans la version 3.2.

tearDownClass()

Méthode de classe appelée après l'exécution des tests de la classe en question. `tearDownClass` est appelée avec la classe comme seul argument et doit être décorée comme une *classmethod()* :

```
@classmethod
def tearDownClass(cls):
    ...
```

Voir *Class and Module Fixtures* pour plus de détails.

Nouveau dans la version 3.2.

run(result=None)

Exécute le test, en collectant le résultat dans l'objet *TestResult* passé comme *result*. Si *result* est omis ou vaut *None*, un objet temporaire de résultat est créé (en appelant la méthode *defaultTestResult()*) et utilisé. L'objet résultat est renvoyé à l'appelant de *run()*.

Le même effet peut être obtenu en appelant simplement l'instance *TestCase*.

Modifié dans la version 3.3 : Les versions précédentes de *run* ne renvoyaient pas le résultat. Pas plus que l'appel d'une instance.

skipTest(reason)

Appeler cette fonction pendant l'exécution d'une méthode de test ou de *setUp()* permet d'ignorer le test en cours. Voir *Ignorer des tests et des erreurs prévisibles* pour plus d'informations.

Nouveau dans la version 3.1.

subTest(msg=None, **params)

Renvoie un gestionnaire de contexte qui exécute le bloc de code du contexte comme un sous-test. *msg* et *params* sont des valeurs optionnelles et arbitraires qui sont affichées chaque fois qu'un sous-test échoue, permettant de les identifier clairement.

Un scénario de test peut contenir un nombre quelconque de déclarations de sous-test, et elles peuvent être imbriquées librement.

Voir *Distinguer les itérations de test à l'aide de sous-tests* pour plus d'informations.

Nouveau dans la version 3.4.

debug()

Lance le test sans collecter le résultat. Ceci permet aux exceptions levées par le test d'être propagées à l'appelant, et donc peut être utilisé pour exécuter des tests sous un débogueur.

La classe *TestCase* fournit plusieurs méthodes d'assertion pour vérifier et signaler les échecs. Le tableau suivant énumère les méthodes les plus couramment utilisées (voir les tableaux ci-dessous pour plus de méthodes d'assertion) :

Méthode	Vérifie que	Disponible en
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2

Toutes les méthodes `assert` prennent en charge un argument `msg` qui, s'il est spécifié, est utilisé comme message d'erreur en cas d'échec (voir aussi `longMessage`). Notez que l'argument mot-clé `msg` peut être passé à `assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, `assertWarnsRegex()`, seulement quand elles sont utilisées comme gestionnaire de contexte.

assertEqual (*first, second, msg=None*)

Vérifie que *first* et *second* sont égaux. Si les valeurs ne sont pas égales, le test échouera.

En outre, si *first* et *second* ont exactement le même type et sont de type *liste*, *tuple*, *dict*, *set*, *frozenset* ou *str* ou tout autre type de sous classe enregistrée dans `addTypeEqualityFunc()`. La fonction égalité spécifique au type sera appelée pour générer une erreur plus utile (voir aussi *liste des méthodes spécifiques de type*).

Modifié dans la version 3.1 : Ajout de l'appel automatique de la fonction d'égalité spécifique au type.

Modifié dans la version 3.2 : Ajout de `assertMultiLineEqual()` comme fonction d'égalité de type par défaut pour comparer les chaînes.

assertNotEqual (*first, second, msg=None*)

Vérifie que *first* et *second* ne sont pas égaux. Si les valeurs sont égales, le test échouera.

assertTrue (*expr, msg=None*)

assertFalse (*expr, msg=None*)

Vérifie que *expr* est vraie (ou fausse).

Notez que cela revient à utiliser `bool(expr) is True` et non à `expr is True` (utilisez `assertIs(expr, True)` pour cette dernière). Cette méthode doit également être évitée lorsque des méthodes plus spécifiques sont disponibles (par exemple `assertEqual(a, b)` au lieu de `assertTrue(a == b)`), car elles fournissent un meilleur message d'erreur en cas d'échec.

assertIs (*first, second, msg=None*)

assertIsNot (*first, second, msg=None*)

Vérifie que *first* et *second* évaluent (ou n'évaluent pas) le même objet.

Nouveau dans la version 3.1.

assertIsNone (*expr, msg=None*)

assertIsNotNone (*expr, msg=None*)

Vérifie que *expr* est (ou n'est pas) la valeur `None`.

Nouveau dans la version 3.1.

assertIn (*first, second, msg=None*)

assertNotIn (*first, second, msg=None*)

Vérifie que *first* est (ou n'est pas) dans *second*.

Nouveau dans la version 3.1.

assertIsInstance (*obj, cls, msg=None*)

assertNotIsInstance (*obj, cls, msg=None*)

Vérifie que *obj* est (ou n'est pas) une instance de *cls* (Ce qui peut être une classe ou un tuple de classes, comme utilisée par `isinstance()`). Pour vérifier le type exact, utilisez `assertIs(type(obj), cls)`.

Nouveau dans la version 3.2.

Il est également possible de vérifier la production des exceptions, des avertissements et des messages de journaux à l'aide des méthodes suivantes :

Méthode	Vérifie que	Dispo- nible en
<code>assertRaises(exc, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> lève bien l'exception <i>exc</i>	
<code>assertRaisesRegex(exc, r, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> lève bien l'exception <i>exc</i> et que le message correspond au motif de l'expression régulière <i>r</i>	3.1
<code>assertWarns(warn, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> lève bien l'avertissement <i>warn</i>	3.2
<code>assertWarnsRegex(warn, r, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> lève bien l'avertissement <i>warn</i> et que le message correspond au motif de l'expression régulière <i>r</i>	3.2
<code>assertLogs(logger, level)</code>	Le bloc <code>with</code> écrit dans le <i>logger</i> avec un niveau minimum égal à <i>level</i>	3.4

assertRaises (*exception*, *callable*, **args*, ***kwargs*)

assertRaises (*exception*, *, *msg*=None)

Vérifie qu'une exception est levée lorsque *callable* est appelé avec n'importe quel argument positionnel ou mot-clé qui est également passé à `assertRaises()`. Le test réussit si *exception* est levée, est en erreur si une autre exception est levée, ou en échec si aucune exception n'est levée. Pour capturer une exception d'un groupe d'exceptions, un couple contenant les classes d'exceptions peut être passé à *exception*.

Si seuls les arguments *exception* et éventuellement *msg* sont donnés, renvoie un gestionnaire de contexte pour que le code sous test puisse être écrit en ligne plutôt que comme une fonction :

```
with self.assertRaises(SomeException):
    do_something()
```

Lorsqu'il est utilisé comme gestionnaire de contexte, `assertRaises()` accepte l'argument de mot-clé supplémentaire *msg*.

Le gestionnaire de contexte enregistre l'exception capturée dans son attribut *exception*. Ceci est particulièrement utile si l'intention est d'effectuer des contrôles supplémentaires sur l'exception levée :

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

Modifié dans la version 3.1 : Ajout de la possibilité d'utiliser `assertRaises()` comme gestionnaire de contexte.

Modifié dans la version 3.2 : Ajout de l'attribut *exception*.

Modifié dans la version 3.3 : Ajout de l'argument *msg* comme mot-clé lorsqu'il est utilisé comme gestionnaire de contexte.

assertRaisesRegex (*exception*, *regex*, *callable*, **args*, ***kwargs*)

assertRaisesRegex (*exception*, *regex*, *, *msg*=None)

Comme `assertRaises()` mais vérifie aussi que *regex* correspond à la représentation de la chaîne de caractères de l'exception levée. *regex* peut être un objet d'expression rationnelle ou une chaîne contenant une expression rationnelle appropriée pour être utilisée par `re.search()`. Exemples :

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                        int, 'XYZ')
```

ou :

```
with self.assertRaisesRegex(ValueError, 'literal'):
    int('XYZ')
```

Nouveau dans la version 3.1 : under the name `assertRaisesRegexp`.

Modifié dans la version 3.2 : Renommé en `assertRaisesRegex()`.

Modifié dans la version 3.3 : Ajout de l'argument `msg` comme mot-clé lorsqu'il est utilisé comme gestionnaire de contexte.

assertWarns (*warning*, *callable*, **args*, ***kws*)

assertWarns (*warning*, *, *msg=None*)

Test qu'un avertissement est déclenché lorsque *callable* est appelé avec n'importe quel argument positionnel ou mot-clé qui est également passé à `assertWarns()`. Le test passe si *warning* est déclenché et échoue s'il ne l'est pas. Toute exception est une erreur. Pour capturer un avertissement dans un ensemble d'avertissements, un couple contenant les classes d'avertissement peut être passé à `warnings`.

Si seuls les arguments * *warning** et éventuellement *msg* sont donnés, renvoie un gestionnaire de contexte pour que le code testé puisse être écrit en ligne plutôt que comme une fonction :

```
with self.assertWarns(SomeWarning):
    do_something()
```

Lorsqu'il est utilisé comme gestionnaire de contexte, `assertWarns()` accepte l'argument de mot-clé supplémentaire *msg*.

Le gestionnaire de contexte stocke l'avertissement capturé dans son attribut `warning`, et la ligne source qui a déclenché les avertissements dans les attributs `filename` et `lineno`. Cette fonction peut être utile si l'intention est d'effectuer des contrôles supplémentaires sur l'avertissement capturé :

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

Cette méthode fonctionne indépendamment des filtres d'avertissement en place lorsqu'elle est appelée.

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : Ajout de l'argument *msg* comme mot-clé lorsqu'il est utilisé comme gestionnaire de contexte.

assertWarnsRegex (*warning*, *regex*, *callable*, **args*, ***kws*)

assertWarnsRegex (*warning*, *regex*, *, *msg=None*)

Comme `assertWarns()` mais vérifie aussi qu'une *regex* corresponde au message de l'avertissement. *regex* peut être un objet d'expression régulière ou une chaîne contenant une expression régulière appropriée pour être utilisée par `re.search()`. Exemple :

```
self.assertWarnsRegex(DeprecationWarning,
    r'legacy_function\(\) is deprecated',
    legacy_function, 'XYZ')
```

ou :

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : Ajout de l'argument *msg* comme mot-clé lorsqu'il est utilisé comme gestionnaire de contexte.

assertLogs (*logger=None*, *level=None*)

Un gestionnaire de contexte pour tester qu'au moins un message est enregistré sur le *logger* ou un de ses enfants, avec au moins le *niveau* donné.

Si donné, *logger* doit être une classe `logging.Logger` objet ou une classe `str` donnant le nom d'un journal. La valeur par défaut est le journal racine *root*, qui capture tous les messages.

S'il est donné, *level* doit être soit un entier, soit son équivalent sous forme de chaîne (par exemple "ERROR" ou `logging.ERROR`). La valeur par défaut est `logging.INFO`.

Le test passe si au moins un message émis à l'intérieur du bloc `with` correspond aux conditions *logger* et *level*, sinon il échoue.

L'objet retourné par le gestionnaire de contexte est une aide à l'enregistrement qui garde la trace des messages de journal correspondants. Il a deux attributs :

records

Une liste d'objets `logging.LogRecord` de messages de log correspondants.

output

Une liste d'objets `str` avec la sortie formatée des messages correspondants.

Exemple :

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

Nouveau dans la version 3.4.

Il existe également d'autres méthodes utilisées pour effectuer des contrôles plus spécifiques, telles que :

Méthode	Vérifie que	Disponible en
<code>assertAlmostEqual(a, b)</code>	<code>round(a-b, 7) == 0</code>	
<code>assertNotAlmostEqual(a, b)</code>	<code>round(a-b, 7) != 0</code>	
<code>assertGreater(a, b)</code>	<code>a > b</code>	3.1
<code>assertGreaterEqual(a, b)</code>	<code>a >= b</code>	3.1
<code>assertLess(a, b)</code>	<code>a < b</code>	3.1
<code>assertLessEqual(a, b)</code>	<code>a <= b</code>	3.1
<code>assertRegex(s, r)</code>	<code>r.search(s)</code>	3.1
<code>assertNotRegex(s, r)</code>	<code>not r.search(s)</code>	3.2
<code>assertCountEqual(a, b)</code>	<i>a</i> and <i>b</i> have the same elements in the same number, regardless of their order	3.2

assertAlmostEqual (*first*, *second*, *places*=7, *msg*=None, *delta*=None)

assertNotAlmostEqual (*first*, *second*, *places*=7, *msg*=None, *delta*=None)

Vérifie que *first* et *second* sont approximativement (ou pas approximativement) égaux en calculant la différence, en arrondissant au nombre donné de décimales *places* (par défaut 7), et en comparant à zéro. Notez que ces méthodes arrondissent les valeurs au nombre donné de *décimales* (par exemple comme la fonction `round()`) et non aux *chiffres significatifs*.

Si *delta* est fourni au lieu de *places*, la différence entre *first* et *second* doit être inférieure ou égale (ou supérieure) à *delta*.

Supplying both *delta* and *places* raises a `TypeError`.

Modifié dans la version 3.2 : `assertAlmostEqual()` considère automatiquement des objets presque égaux qui se comparent égaux. `assertNotAlmostEqual()` échoue automatiquement si les objets qui se comparent sont égaux. Ajout de l'argument mot-clé *delta*.

assertGreater (*first*, *second*, *msg*=None)

assertGreaterEqual (*first*, *second*, *msg*=None)

assertLess (*first, second, msg=None*)

assertLessEqual (*first, second, msg=None*)

Vérifie que *first* est respectivement $>$, $>=$, $>=$, $<$ ou $<=$ à *second* selon le nom de la méthode. Sinon, le test échouera :

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

Nouveau dans la version 3.1.

assertRegex (*text, regex, msg=None*)

assertNotRegex (*text, regex, msg=None*)

Vérifie qu'une recherche par motif *regex* correspond (ou ne correspond pas) à *text*. En cas d'échec, le message d'erreur inclura le motif et le *texte* (ou le motif et la partie du *texte* qui correspond de manière inattendue). *regex* peut être un objet d'expression régulière ou une chaîne contenant une expression régulière appropriée pour être utilisée par `re.search()`.

Nouveau dans la version 3.1 : under the name `assertRegexpMatches`.

Modifié dans la version 3.2 : La méthode `assertRegexpMatches()` a été renommé en `assertRegex()`.

Nouveau dans la version 3.2 : `assertNotRegex()`.

Nouveau dans la version 3.5 : Le nom `assertNotRegexpMatches` est un alias obsolète pour `assertNotRegex()`.

assertCountEqual (*first, second, msg=None*)

Vérifie que la séquence *first* contient les mêmes éléments que *second*, quel que soit leur ordre. Si ce n'est pas le cas, un message d'erreur indiquant les différences entre les séquences est généré.

Les éléments en double ne sont *pas* ignorés lors de la comparaison entre *first* et *second*. Il vérifie si chaque élément a le même nombre dans les deux séquences. Équivalent à : `assertEqual(Counter(list(first)), Counter(list(second)))` mais fonctionne aussi avec des séquences d'objets non *hachables*.

Nouveau dans la version 3.2.

La méthode `assertEqual()` envoie le contrôle d'égalité pour les objets du même type à différentes méthodes spécifiques au type. Ces méthodes sont déjà implémentées pour la plupart des types intégrés, mais il est également possible d'enregistrer de nouvelles méthodes en utilisant `addTypeEqualityFunc()` :

addTypeEqualityFunc (*typeobj, function*)

Enregistre une méthode spécifique appelée par `assertEqual()` pour vérifier si deux objets exactement du même *typeobj* (et non leurs sous-classes) sont égaux. *function* doit prendre deux arguments positionnels et un troisième argument mot-clé *msg=None* tout comme `assertEqual()` le fait. Il doit lever `self.failureException(msg)` lorsqu'une inégalité entre les deux premiers paramètres est détectée en fournissant éventuellement des informations utiles et expliquant l'inégalité en détail dans le message d'erreur.

Nouveau dans la version 3.1.

La liste des méthodes spécifiques utilisées automatiquement par `assertEqual()` est résumée dans le tableau suivant. Notez qu'il n'est généralement pas nécessaire d'invoquer ces méthodes directement.

Méthode	Utilisé pour comparer	Disponible en
<code>assertMultiLineEqual(a, b)</code>	chaînes	3.1
<code>assertSequenceEqual(a, b)</code>	séquences	3.1
<code>assertListEqual(a, b)</code>	listes	3.1
<code>assertTupleEqual(a, b)</code>	n-uplets	3.1
<code>assertSetEqual(a, b)</code>	sets ou frozensets	3.1
<code>assertDictEqual(a, b)</code>	dictionnaires	3.1

assertMultiLineEqual (*first, second, msg=None*)

Vérifie que la chaîne sur plusieurs lignes *first* est égale à la chaîne *second*. Si les deux chaînes de caractères ne sont pas égales, un *diff* mettant en évidence les différences est inclus dans le message d'erreur. Cette méthode est utilisée par défaut pour comparer les chaînes avec `assertEqual()`.

Nouveau dans la version 3.1.

assertSequenceEqual (*first, second, msg=None, seq_type=None*)

Vérifie que deux séquences sont égales. Si un *seq_type* est fourni, *first* et *second* doivent tous deux être des instances de *seq_type* ou un échec est levé. Si les séquences sont différentes, un message d'erreur indiquant la différence entre les deux est généré.

Cette méthode n'est pas appelée directement par `assertEqual()`, mais sert à implémenter `assertListEqual()` et `assertTupleEqual()`.

Nouveau dans la version 3.1.

assertListEqual (*first, second, msg=None*)

assertTupleEqual (*first, second, msg=None*)

Vérifie que deux listes ou deux n-uplets sont égaux. Si ce n'est pas le cas, un message d'erreur qui ne montre que les différences entre les deux est généré. Une erreur est également signalée si l'un ou l'autre des paramètres n'est pas du bon type. Ces méthodes sont utilisées par défaut pour comparer des listes ou des couples avec `assertEqual()`.

Nouveau dans la version 3.1.

assertSetEqual (*first, second, msg=None*)

Vérifie que deux ensembles sont égaux. Si ce n'est pas le cas, un message d'erreur s'affiche et indique les différences entre les *sets*. Cette méthode est utilisée par défaut lors de la comparaison de *sets* ou de *frozensets* avec `assertEqual()`.

Échoue si l'un des objets *first* ou *second* n'a pas de méthode `set.difference()`.

Nouveau dans la version 3.1.

assertDictEqual (*first, second, msg=None*)

Vérifie que deux dictionnaires sont égaux. Si ce n'est pas le cas, un message d'erreur qui montre les différences dans les dictionnaires est généré. Cette méthode est utilisée par défaut pour comparer les dictionnaires dans les appels à `assertEqual()`.

Nouveau dans la version 3.1.

Enfin, la classe `TestCase` fournit les méthodes et attributs suivants :

fail (*msg=None*)

Indique un échec du test sans condition, avec *msg* ou `None` pour le message d'erreur.

failureException

Cet attribut de classe donne l'exception levée par la méthode de test. Si un *framework* de tests doit utiliser une exception spécialisée, probablement pour enrichir l'exception d'informations additionnels., il doit hériter de cette classe d'exception pour *bien fonctionner* avec le *framework*. La valeur initiale de cet attribut est `AssertionError`.

longMessage

Cet attribut de classe détermine ce qui se passe lorsqu'un message d'échec personnalisé est passé en argument au paramètre *msg* à un appel `assertXXXX` qui échoue. `True` est la valeur par défaut. Dans ce cas, le message personnalisé est ajouté à la fin du message d'erreur standard. Lorsqu'il est réglé sur `False`, le message personnalisé remplace le message standard.

Le paramétrage de la classe peut être écrasé dans les méthodes de test individuelles en assignant un attribut d'instance, `self.longMessage`, à `True` ou `False` avant d'appeler les méthodes d'assertion.

Le réglage de la classe est réinitialisé avant chaque appel de test.

Nouveau dans la version 3.1.

maxDiff

Cet attribut contrôle la longueur maximale des *diffs* en sortie des méthodes qui génèrent des *diffs* en cas d'échec. La valeur par défaut est 80*8 caractères. Les méthodes d'assertions affectées par cet attribut sont `assertSequenceEqual()` (y compris toutes les méthodes de comparaison de séquences qui lui sont déléguées), `assertDictEqual()` et `assertMultiLineEqual()`.

Régler `maxDiff` sur `None` signifie qu'il n'y a pas de longueur maximale pour les *diffs*.

Nouveau dans la version 3.2.

Les *frameworks* de test peuvent utiliser les méthodes suivantes pour recueillir des informations sur le test :

countTestCases ()

Renvoie le nombre de tests représentés par cet objet test. Pour les instances de *TestCase*, c'est toujours 1.

defaultTestResult ()

Retourne une instance de la classe de résultat de test qui doit être utilisée pour cette classe de cas de test (si aucune autre instance de résultat n'est fournie à la méthode *run ()*).

Pour les instances de *TestCase*, c'est toujours une instance de *TestResult*; les sous-classes de *TestCase* peuvent la remplacer au besoin.

id ()

Retourne une chaîne identifiant le cas de test spécifique. Il s'agit généralement du nom complet de la méthode de test, y compris le nom du module et de la classe.

shortDescription ()

Renvoie une description du test, ou *None* si aucune description n'a été fournie. L'implémentation par défaut de cette méthode renvoie la première ligne de la *docstring* de la méthode de test, si disponible, ou *None*.

Modifié dans la version 3.1 : En 3.1, ceci a été modifié pour ajouter le nom du test à la description courte, même en présence d'une *docstring*. Cela a causé des problèmes de compatibilité avec les extensions *unittest* et l'ajout du nom du test a été déplacé dans la classe *TextTestResult* dans Python 3.2.

addCleanup (function, *args, **kwargs)

Ajout d'une fonction à appeler après *tearDown ()* pour nettoyer les ressources utilisées pendant le test. Les fonctions seront appelées dans l'ordre inverse de l'ordre dans lequel elles ont été ajoutées (LIFO). Elles sont appelées avec tous les arguments et arguments de mots-clés passés à *addCleanup ()* quand elles sont ajoutées.

Si *setUp ()* échoue, cela signifie que *tearDown ()* n'est pas appelé, alors que les fonctions de nettoyage ajoutées seront toujours appelées.

Nouveau dans la version 3.1.

doCleanups ()

Cette méthode est appelée sans conditions après *tearDown ()*, ou après *setUp ()* si *setUp ()* lève une exception.

Cette méthode est chargée d'appeler toutes les fonctions de nettoyage ajoutées par *addCleanup ()*. Si vous avez besoin de fonctions de nettoyage à appeler avant l'appel à *tearDown ()* alors vous pouvez appeler *doCleanups ()* vous-même.

doCleanups () extrait les méthodes de la pile des fonctions de nettoyage une à la fois, de sorte qu'elles peuvent être appelées à tout moment.

Nouveau dans la version 3.1.

class unittest.FunctionTestCase (testFunc, setUp=None, tearDown=None, description=None)

Cette classe implémente la partie de l'interface *TestCase* qui permet au lanceur de test de piloter le scénario de test, mais ne fournit pas les méthodes que le code test peut utiliser pour vérifier et signaler les erreurs. Ceci est utilisé pour créer des scénarios de test utilisant du code de test existant afin de faciliter l'intégration dans un *framework* de test basé sur *unittest*.

Alias obsolètes

Pour des raisons historiques, certaines méthodes de la classe `TestCase` avaient un ou plusieurs alias qui sont maintenant obsolètes. Le tableau suivant énumère les noms corrects ainsi que leurs alias obsolètes :

Nom de méthode	Alias obsolètes	Alias obsolètes
<code>assertEqual()</code>	<code>failUnlessEqual</code>	<code>assertEquals</code>
<code>assertNotEqual()</code>	<code>failIfEqual</code>	<code>assertNotEquals</code>
<code>assertTrue()</code>	<code>failUnless</code>	<code>assert_</code>
<code>assertFalse()</code>	<code>failIf</code>	
<code>assertRaises()</code>	<code>failUnlessRaises</code>	
<code>assertAlmostEqual()</code>	<code>failUnlessAlmostEqual</code>	<code>assertAlmostEquals</code>
<code>assertNotAlmostEqual()</code>	<code>failIfAlmostEqual</code>	<code>assertNotAlmostEquals</code>
<code>assertRegex()</code>		<code>assertRegexpMatches</code>
<code>assertNotRegex()</code>		<code>assertNotRegexpMatches</code>
<code>assertRaisesRegex()</code>		<code>assertRaisesRegexp</code>

Obsolète depuis la version 3.1 : the `fail*` aliases listed in the second column.

Obsolète depuis la version 3.2 : the `assert*` aliases listed in the third column.

Obsolète depuis la version 3.2 : Les expressions `assertRegexpMatches` et `assertRaisesRegexp` ont été renommées en `assertRegex()` et `assertRaisesRegex()`.

Obsolète depuis la version 3.5 : `assertNotRegexpMatches` en faveur de `assertNotRegex()`.

Regroupement des tests

class `unittest.TestSuite` (*tests=()*)

Cette classe représente une agrégation de cas de test individuels et de suites de tests. La classe présente l'interface requise par le lanceur de test pour être exécutée comme tout autre cas de test. L'exécution d'une instance de `TestSuite` est identique à l'itération sur la suite, en exécutant chaque test indépendamment.

Si *tests* est fourni, il doit s'agir d'un itérable de cas de test individuels ou d'autres suites de test qui seront utilisés pour construire la suite initial. Des méthodes supplémentaires sont fournies pour ajouter ultérieurement des cas de test et des suites à la collection.

Les objets `TestSuite` se comportent comme les objets `TestCase`, sauf qu'ils n'implémentent pas réellement un test. Au lieu de cela, ils sont utilisés pour regrouper les tests en groupes de tests qui doivent être exécutés ensemble. Des méthodes supplémentaires sont disponibles pour ajouter des tests aux instances de `TestSuite` :

addTest (*test*)

Ajouter un objet `TestCase` ou `TestSuite` à la suite de tests.

addTests (*tests*)

Ajouter tous les tests d'un itérable d'instances de `TestCase` et de `TestSuite` à cette suite de tests.

C'est l'équivalent d'une itération sur *tests*, appelant `addTest()` pour chaque élément.

`TestSuite` partage les méthodes suivantes avec `TestCase` :

run (*result*)

Exécute les tests associés à cette suite, en collectant le résultat dans l'objet de résultat de test passé par *result*. Remarque que contrairement à `TestCase.run()`, `TestSuite.run()` nécessite que l'objet résultat soit passé.

debug ()

Exécute les tests associés à cette suite sans collecter le résultat. Ceci permet aux exceptions levées par le test d'être propagées à l'appelant et peut être utilisé pour exécuter des tests sous un débogueur.

countTestCases ()

Renvoie le nombre de tests représentés par cet objet de test, y compris tous les tests individuels et les sous-suites.

__iter__()

Les tests groupés par une classe `TestSuite` sont toujours accessibles par itération. Les sous-classes peuvent fournir paresseusement des tests en surchargeant `__iter__()`. Notez que cette méthode peut être appelée plusieurs fois sur une même suite (par exemple lors du comptage des tests ou de la comparaison pour l'égalité) et que les tests retournés par itérations répétées avant `TestSuite.run()` doivent être les mêmes pour chaque itération. Après `TestSuite.run()`, les appelants ne devraient pas se fier aux tests retournés par cette méthode à moins qu'ils n'utilisent une sous-classe qui remplace `TestSuite._removeTestAtIndex()` pour préserver les références des tests.

Modifié dans la version 3.2 : Dans les versions précédentes, la classe `TestSuite` accédait aux tests directement plutôt que par itération, donc surcharger la méthode `__iter__()` n'était pas suffisante pour fournir les tests.

Modifié dans la version 3.4 : Dans les versions précédentes, la classe `TestSuite` contenait des références à chaque `TestCase` après l'appel à `TestSuite.run()`. Les sous-classes peuvent restaurer ce comportement en surchargeant `TestSuite._removeTestAtIndex()`.

Dans l'utilisation typique de l'objet `TestSuite`, la méthode `run()` est invoquée par une classe `TestRunner` plutôt que par le système de test de l'utilisateur.

Chargement et exécution des tests

class `unittest.TestLoader`

La classe `TestLoader` est utilisée pour créer des suites de tests à partir de classes et de modules. Normalement, il n'est pas nécessaire de créer une instance de cette classe ; le module `unittest` fournit une instance qui peut être partagée comme `unittest.defaultTestLoader`. L'utilisation d'une sous-classe ou d'une instance permet cependant de personnaliser certaines propriétés configurables.

Les objets de la classe `TestLoader` ont les attributs suivants :

errors

Une liste des erreurs non fatales rencontrées lors du chargement des tests. Il est impossible de faire une remise à zéro pendant le chargement. Les erreurs fatales sont signalées par la méthode correspondante qui lève une exception à l'appelant. Les erreurs non fatales sont également indiquées par un test synthétique qui lève l'erreur initiale lors de l'exécution.

Nouveau dans la version 3.5.

Les objets de la classe `TestLoader` ont les attributs suivants :

loadTestsFromTestCase (*testCaseClass*)

Renvoie une suite de tous les cas de test contenus dans la classe `TestCaseClass` dérivée de `TestCase`.

Une instance de cas de test est créée pour chaque méthode nommée par `getTestCaseNames()`. Par défaut, ce sont les noms des méthodes commençant par « test ». Si `getTestTestCaseNames()` ne renvoie aucune méthode, mais que la méthode `runTest()` est implémentée, un seul cas de test est créé pour cette méthode à la place.

loadTestsFromModule (*module*, *pattern=None*)

Renvoie une suite de tous les cas de test contenus dans le module donné. Cette méthode recherche *module* pour les classes dérivées de `TestCase` et crée une instance de la classe pour chaque méthode de test définie pour cette classe.

Note : Bien que l'utilisation d'une hiérarchie de classes `TestCase` (les classes dérivées de `TestCase`) peut être un moyen pratique de partager des *fixtures* et des fonctions utilitaires, définir une méthode de test pour des classes de base non destinées à être directement instanciées ne marche pas bien avec cette méthode. Cela peut toutefois s'avérer utile lorsque les *fixtures* sont différentes et définies dans des sous-classes.

Si un module fournit une fonction `load_tests`, il est appelé pour charger les tests. Cela permet aux modules de personnaliser le chargement des tests. C'est le protocole *load_tests protocol*. L'argument *pattern* est passé comme troisième argument à `load_tests`.

Modifié dans la version 3.2 : Ajout de la prise en charge de `load_tests`.

Modifié dans la version 3.5 : L'argument par défaut non documenté et non officiel `use_load_tests` est déprécié et ignoré, bien qu'il soit toujours accepté pour la compatibilité descendante. La méthode accepte aussi maintenant un argument `pattern` qui est passé à `load_tests` comme troisième argument.

loadTestsFromName (*name*, *module=None*)

Renvoie une suite de tous les cas de test en fonction d'un spécificateur de chaîne de caractères.

Le spécificateur *name* est un « nom pointillé » qui peut être résolu soit par un module, une classe de cas de test, une méthode de test dans une classe de cas de test, une instance de `TestSuite`, ou un objet appellable qui retourne une instance de classe `TestCase` ou de classe `TestSuite`. Ces contrôles sont appliqués dans l'ordre indiqué ici, c'est-à-dire qu'une méthode sur une classe de cas de test possible sera choisie comme « méthode de test dans une classe de cas de test », plutôt que comme « un objet appellable ».

Par exemple, si vous avez un module `SampleTests` contenant une classe `TestCase` (classe dérivée de la classe `SampleTestCase`) avec trois méthodes de test (`test_one()`, `test_two()` et `test_three()`), l'élément spécificateur `SampleTests.sampleTestCase` renvoie une suite qui va exécuter les trois méthodes de tests. L'utilisation du spécificateur `SampleTests.SampleTestCase.test_two` permettrait de retourner une suite de tests qui ne lancerait que la méthode `test_two()`. Le spécificateur peut se référer à des modules et packages qui n'ont pas été importés. Ils seront importés par un effet de bord.

La méthode résout facultativement *name* relatif au *module* donné.

Modifié dans la version 3.5 : Si une `ImportError` ou `AttributeError` se produit pendant la traversée de *name*, un test synthétique qui enrichie l'erreur produite lors de l'exécution est renvoyé. Ces erreurs sont incluses dans les erreurs accumulées par `self.errors`.

loadTestsFromNames (*names*, *module=None*)

Similaire à `loadTestsFromName()`, mais prend une séquence de noms plutôt qu'un seul nom. La valeur renvoyée est une suite de tests qui gère tous les tests définis pour chaque nom.

getTestCaseNames (*testCaseClass*)

Renvoie une séquence triée de noms de méthodes trouvés dans *testCaseClass*; ceci doit être une sous-classe de `TestCase`.

discover (*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

Trouve tous les modules de test en parcourant les sous-répertoires du répertoire de démarrage spécifié, et renvoie un objet `TestSuite` qui les contient. Seuls les fichiers de test qui correspondent à *pattern* sont chargés. Seuls les noms de modules qui sont importables (c'est-à-dire qui sont des identifiants Python valides) sont chargés.

Tous les modules de test doivent être importables depuis la racine du projet. Si le répertoire de démarrage n'est pas la racine, le répertoire racine doit être spécifié séparément.

Si l'importation d'un module échoue, par exemple en raison d'une erreur de syntaxe, celle-ci est alors enregistrée comme une erreur unique et la découverte se poursuit. Si l'échec de l'importation est dû au fait que `SkipTest` est levé, il est enregistré comme un saut plutôt que comme un message d'erreur.

Si un paquet (un répertoire contenant un fichier nommé `__init__.py`) est trouvé, le paquet est alors vérifié pour une fonction `load_tests`. Si elle existe, elle s'appellera `package.load_tests(loader, tests, pattern)`. Le mécanisme de découverte de test prend soin de s'assurer qu'un paquet n'est vérifié qu'une seule fois au cours d'une invocation, même si la fonction `load_tests` appelle elle-même `loader.discover`.

Si `load_tests` existe alors la découverte ne poursuit pas la récursion dans le paquet, `load_tests` a la responsabilité de charger tous les tests dans le paquet.

Le motif n'est délibérément pas stocké en tant qu'attribut du chargeur afin que les paquets puissent continuer à être découverts eux-mêmes. *top_level_dir* est stocké de sorte que `load_tests` n'a pas besoin de passer cet argument à `loader.discover()`.

start_dir peut être un nom de module ainsi qu'un répertoire.

Nouveau dans la version 3.2.

Modifié dans la version 3.4 : Les modules qui lèvent `SkipTest` lors de l'importation sont enregistrés comme des sauts et non des erreurs. Le mécanisme de découverte fonctionne pour les *paquets-espaces de nommage*. Les chemins sont triés avant d'être importés pour que l'ordre d'exécution soit le même, même si l'ordre du système de fichiers sous-jacent ne dépend pas du nom du fichier.

Modifié dans la version 3.5 : Les paquets trouvés sont maintenant vérifiés pour `load_tests` indépendamment du fait que leur chemin d'accès corresponde ou non à *pattern*, car il est impossible pour un nom de paquet de correspondre au motif par défaut.

Les attributs suivants d'une classe `TestLoader` peuvent être configurés soit par héritage, soit par affectation sur une instance :

testMethodPrefix

Chaîne donnant le préfixe des noms de méthodes qui seront interprétés comme méthodes de test. La valeur par défaut est `'test'`.

Ceci affecte les méthodes `getTestCaseNames()` et toutes les méthodes `loadTestsFrom*()`.

sortTestMethodsUsing

Fonction à utiliser pour comparer les noms de méthodes lors de leur tri dans les méthodes `getTestCaseNames()` et toutes les méthodes `loadTestsFrom*()`.

suiteClass

Objet callable qui construit une suite de tests à partir d'une liste de tests. Aucune méthode sur l'objet résultant n'est nécessaire. La valeur par défaut est la classe `TestSuite`.

Cela affecte toutes les méthodes `loadTestsFrom*()`.

class unittest.TestResult

Cette classe est utilisée pour compiler des informations sur les tests qui ont réussi et ceux qui ont échoué.

Un objet `TestResult` stocke les résultats d'un ensemble de tests. Les classes `TestCase` et `TestSuite` s'assurent que les résultats sont correctement enregistrés. Les auteurs du test n'ont pas à se soucier de l'enregistrement des résultats des tests.

Les cadriciels de test construits sur `unittest` peuvent nécessiter l'accès à l'objet `TestResult` généré en exécutant un ensemble de tests à des fins de génération de comptes-rendu. Une instance de `TestResult` est alors renvoyée par la méthode `TestRunner.run()` à cette fin.

Les instances de `TestResult` ont les attributs suivants qui sont intéressants pour l'inspection des résultats de l'exécution d'un ensemble de tests :

errors

Une liste contenant un couple d'instances de `TestCase` et une chaînes de caractères contenant des traces formatées. Chaque couple représente un test qui a levé une exception inattendue.

failures

Une liste contenant un couple d'instances de `TestCase` et une chaînes de caractères contenant des traces formatées. Chaque tuple représente un test où un échec a été explicitement signalé en utilisant les méthodes `TestCase.assert*()`.

skipped

Une liste contenant un couple d'instances de `TestCase` et une chaînes de caractères contenant la raison de l'omission du test.

Nouveau dans la version 3.1.

expectedFailures

Une liste contenant un couple d'instance `TestCase` et une chaînes de caractères contenant des traces formatées. Chaque couple représente un échec attendu du scénario de test.

unexpectedSuccesses

Une liste contenant les instances `TestCase` qui ont été marquées comme des échecs attendus, mais qui ont réussi.

shouldStop

A positionner sur `True` quand l'exécution des tests doit être arrêter par `stop()`.

testsRun

Le nombre total de tests effectués jusqu'à présent.

buffer

S'il est défini sur `true`, `sys.stdout` et `sys.stderr` sont mis dans un tampon entre les appels de `startTest()` et `stopTest()`. La sortie collectée est répercutée sur les sorties `sys.stdout` et `sys.stderr` réels uniquement en cas d'échec ou d'erreur du test. Toute sortie est également attachée au message d'erreur.

Nouveau dans la version 3.2.

failfast

Si la valeur est *true* `stop()` est appelée lors de la première défaillance ou erreur, ce qui interrompt le test en cours d'exécution.

Nouveau dans la version 3.2.

tb_locals

Si la valeur est *true*, les variables locales sont affichées dans les traces d'appels.

Nouveau dans la version 3.5.

wasSuccessful()

Renvoie *True* si tous les tests effectués jusqu'à présent ont réussi, sinon renvoie *False*.

Modifié dans la version 3.4 : Renvoie *False* s'il y a eu des *unexpectedSuccesses* dans les tests annotés avec le décorateur *expectedFailure()*.

stop()

Cette méthode peut être appelée pour signaler que l'ensemble des tests en cours d'exécution doit être annulé en définissant l'attribut *shouldStop* sur *True*. Les instances de *TestRunner* doivent respecter ce signal et se terminer sans exécuter de tests supplémentaires.

Par exemple, cette fonctionnalité est utilisée par la classe *TextTestRunner* pour arrêter le cadriciel de test lorsque l'utilisateur lance une interruption clavier. Les outils interactifs qui fournissent des implémentations de *TestRunner* peuvent l'utiliser de la même manière.

Les méthodes suivantes de la classe *TestResult* sont utilisées pour maintenir les structures de données internes, et peuvent être étendues dans des sous-classes pour gérer des exigences supplémentaires en termes de compte-rendu. Cette fonction est particulièrement utile pour créer des outils qui prennent en charge la génération de rapports interactifs pendant l'exécution des tests.

startTest(test)

Appelé lorsque le scénario de test *test* est sur le point d'être exécuté.

stopTest(test)

Appelé après l'exécution du cas de test *test*, quel qu'en soit le résultat.

startTestRun()

Appelé une fois avant l'exécution des tests.

Nouveau dans la version 3.1.

stopTestRun()

Appelé une fois après l'exécution des tests.

Nouveau dans la version 3.1.

addError(test, err)

Appelé lorsque le cas de test *test* soulève une exception inattendue. *err* est un couple du formulaire renvoyé par `sys.exc_info() : (type, valeur, traceback)`.

L'implémentation par défaut ajoute un couple (*test*, *formatted_err*) à l'attribut *errors* de l'instance, où *formatted_err* est une trace formatée à partir de *err*.

addFailure(test, err)

Appelé lorsque le cas de test *test* soulève une exception inattendue. *err* est un triplet de la même forme que celui renvoyé par `sys.exc_info() : (type, valeur, traceback)`.

L'implémentation par défaut ajoute un couple (*test*, *formatted_err*) à l'attribut *errors* de l'instance, où *formatted_err* est une trace formatée à partir de *err*.

addSuccess(test)

Appelé lorsque le scénario de test *test* réussit.

L'implémentation par défaut ne fait rien.

addSkip(test, reason)

Appelé lorsque le scénario de test *test* est ignoré. *raison* est la raison pour laquelle le test donné a été ignoré.

L'implémentation par défaut ajoute un couple (*test*, *raison*) à l'attribut *skipped* de l'instance.

addExpectedFailure(test, err)

Appelé lorsque le scénario de test *test* échoue, mais qui a été marqué avec le décorateur *expectedFailure()*.

L'implémentation par défaut ajoute un couple (`test`, `formatted_err`) à l'attribut `errors` de l'instance, où `formatted_err` est une trace formatée à partir de `err`.

addUnexpectedSuccess (*test*)

Appelé lorsque le scénario de test *test* réussit, mais que ce scénario a été marqué avec le décorateur `expectedFailure()`.

L'implémentation par défaut ajoute le test à l'attribut `unexpectedSuccesses` de l'instance.

addSubTest (*test*, *subtest*, *outcome*)

Appelé à la fin d'un sous-test. *test* est le cas de test correspondant à la méthode de test. *subtest* est une instance dérivée de `TestCase` décrivant le sous-test.

Si *outcome* est `None`, le sous-test a réussi. Sinon, il a échoué avec une exception où *outcome* est un triplet du formulaire renvoyé par `sys.exc_info()` : (type, valeur, traceback).

L'implémentation par défaut ne fait rien lorsque le résultat est un succès, et enregistre les échecs de sous-test comme des échecs normaux.

Nouveau dans la version 3.4.

class `unittest.TextTestResult` (*stream*, *descriptions*, *verbosity*)

Une implémentation concrète de `TestResult` utilisé par la classe `TextTestRunner`.

Nouveau dans la version 3.2 : Cette classe s'appelait auparavant `_TextTestResult`. L'ancien nom existe toujours en tant qu'alias, mais il est obsolète.

`unittest.defaultTestLoader`

Instance de la classe `TestLoader` destinée à être partagée. Si aucune personnalisation de la classe `TestLoader` n'est nécessaire, cette instance peut être utilisée au lieu de créer plusieurs fois de nouvelles instances.

class `unittest.TextTestRunner` (*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, *, *tb_locals=False*)

Une implémentation de base d'un lanceur de test qui fournit les résultats dans un flux. Si *stream* est `None`, par défaut, `sys.stderr` est utilisé comme flux de sortie. Cette classe a quelques paramètres configurables, mais est essentiellement très simple. Les applications graphiques qui exécutent des suites de tests doivent fournir des implémentations alternatives. De telles implémentations doivent accepter `**kwargs` car l'interface pour construire les lanceurs change lorsque des fonctionnalités sont ajoutées à `unittest`.

Par défaut, ce lanceur affiche `DeprecationWarning`, `PendingDeprecationWarning`, `ResourceWarning` et `ImportWarning` même si elles sont *ignorées par défaut*. Les avertissements causés par *des méthodes *unittest* dépréciées* sont également spéciaux et, lorsque les filtres d'avertissement sont `default` ou `always`, ils n'apparaissent qu'une fois par module, afin d'éviter trop de messages d'alerte. Ce comportement peut être annulé en utilisant les options `-Wd` ou `-Wa` de Python (voir Gestion des avertissements) et en laissant *warnings* à `None`.

Modifié dans la version 3.2 : Ajout du paramètre `warnings`.

Modifié dans la version 3.2 : Le flux par défaut est défini sur `sys.stderr` au moment de l'instanciation plutôt qu'à l'importation.

Modifié dans la version 3.5 : Ajout du paramètre `tb_locals`.

_makeResult ()

Cette méthode renvoie l'instance de `TestResult` utilisée par `run()`. Il n'est pas destiné à être appelé directement, mais peut être surchargée dans des sous-classes pour fournir un `TestResult` personnalisé.

`_makeResult()` instancie la classe ou l'appelable passé dans le constructeur `TextTestRunner` comme argument `resultclass`. Il vaut par défaut `TextTestResult` si aucune `resultclass` n'est fournie. La classe de résultat est instanciée avec les arguments suivants :

```
stream, descriptions, verbosity
```

run (*test*)

This method is the main public interface to the `TextTestRunner`. This method takes a `TestSuite` or `TestCase` instance. A `TestResult` is created by calling `_makeResult()` and the test(s) are run and the results printed to stdout.

```
unittest.main(module='__main__', defaultTest=None, argv=None, testRunner=None, testLoader=unittest.defaultTestLoader, exit=True, verbosity=1, failfast=None, catchbreak=None, buffer=None, warnings=None)
```

Un programme en ligne de commande qui charge un ensemble de tests à partir du *module* et les exécute. L'utilisation principale est de rendre les modules de test facilement exécutables. L'utilisation la plus simple pour cette fonction est d'inclure la ligne suivante à la fin d'un script de test :

```
if __name__ == '__main__':
    unittest.main()
```

Vous pouvez exécuter des tests avec des informations plus détaillées en utilisant l'option de verbosité :

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

L'argument *defaultTest* est soit le nom d'un seul test, soit un itérable de noms de test à exécuter si aucun nom de test n'est spécifié via *argv*. Si aucun nom de test n'est fourni via *argv*, tous les tests trouvés dans *module* sont exécutés.

L'argument *argv* peut être une liste d'options passées au programme, le premier élément étant le nom du programme. S'il n'est pas spécifié ou vaut *None*, les valeurs de *sys.argv* sont utilisées.

L'argument *testRunner* peut être soit une classe de lanceur de test, soit une instance déjà créée de celle-ci. Par défaut, *main* appelle *sys.exit()* avec un code de sortie indiquant le succès ou l'échec des tests exécutés.

L'argument *testLoader* doit être une instance de *TestLoader*, et par défaut de *defaultTestLoader*.

Les *main* sont utilisés à partir de l'interpréteur interactif en passant dans l'argument *exit=False*. Ceci affiche le résultat sur la sortie standard sans appeler *sys.exit()* :

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

Les paramètres *failfast*, *catchbreak* et *buffer* ont le même effet que la même option en ligne de commande *command-line options*.

L'argument *warnings* spécifie l'argument *filtre d'avertissement* qui doit être utilisé lors de l'exécution des tests. Si elle n'est pas spécifiée, elle reste réglée sur *None* si une option *-W* est passée à **python** (voir Utilisation des avertissements), sinon elle sera réglée sur *'default'*.

L'appel de *main* renvoie en fait une instance de la classe *TestProgram*. Le résultat des tests effectués est enregistré sous l'attribut *result*.

Modifié dans la version 3.1 : Ajout du paramètre *exit*.

Modifié dans la version 3.2 : Ajout des paramètres *verbosity*, *failfast*, *catchbreak*, *buffer* et *warnings*.

Modifié dans la version 3.4 : Le paramètre *defaultTest* a été modifié pour accepter également un itérable de noms de test.

Protocole de chargement des tests (*load_tests Protocol*)

Nouveau dans la version 3.2.

Les modules ou paquets peuvent personnaliser la façon dont les tests sont chargés à partir de ceux-ci pendant l'exécution des tests ou pendant la découverte de tests en implémentant une fonction appelée *load_tests*.

Si un module de test définit *load_tests* il est appelé par *TestLoader.loadTestsFromModule()* avec les arguments suivants :

```
load_tests(loader, standard_tests, pattern)
```

où *pattern* est passé directement depuis *loadTestsFromModule*. La valeur par défaut est *None*.

Elle doit renvoyer une classe *TestSuite*.

loader est l'instance de *TestLoader* qui effectue le chargement. *standard_tests* sont les tests qui sont chargés par défaut depuis le module. Il est courant que les modules de test veuillent seulement ajouter ou supprimer des tests de l'ensemble standard de tests. Le troisième argument est utilisé lors du chargement de paquets dans le cadre de la découverte de tests.

Une fonction typique de *load_tests* qui charge les tests d'un ensemble spécifique de classes *TestCase* peut ressembler à :

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

Si la découverte est lancée dans un répertoire contenant un paquet, soit à partir de la ligne de commande, soit en appelant *TestLoader.discover()*, alors le système recherche dans le fichier du paquet *__init__.py* la fonction *load_tests*. Si cette fonction n'existe pas, la découverte se poursuit dans le paquet comme si c'était juste un autre répertoire. Sinon, la découverte des tests du paquet est effectuée par *load_tests* qui est appelé avec les arguments suivants :

```
load_tests(loader, standard_tests, pattern)
```

Doit renvoyer une classe *TestSuite* représentant tous les tests du paquet. (*standard_tests* ne contient que les tests collectés dans le fichier *__init__.py*).

Comme le motif est passé à *load_tests*, le paquet est libre de continuer (et potentiellement de modifier) la découverte des tests. Une fonction « ne rien faire » *load_tests* pour un paquet de test ressemblerait à :

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

Modifié dans la version 3.5 : La découverte de test ne vérifie plus que les noms de paquets correspondent à *pattern* en raison de l'impossibilité de trouver des noms de paquets correspondant au motif par défaut.

26.4.9 Classes et modules d'aménagements des tests

Les classes et modules d'aménagements des tests sont implémentés dans *TestSuite*. Lorsque la suite de tests rencontre un test d'une nouvelle classe, alors *tearDownClass()* de la classe précédente (s'il y en a une) est appelé, suivi de *setUpClass()* de la nouvelle classe.

De même, si un test provient d'un module différent du test précédent, alors *tearDownModule* du module précédent est exécuté, suivi par *setUpModule* du nouveau module.

Après que tous les tests ont été exécutés, les *tearDownClass* et *tearDownModule* finaux sont exécutés.

Notez que les aménagements de tests partagés ne fonctionnent pas bien avec de « potentielles » fonctions comme la parallélisation de test et qu'ils brisent l'isolation des tests. Ils doivent être utilisés avec parcimonie.

L'ordre par défaut des tests créés par les chargeurs de tests unitaires est de regrouper tous les tests des mêmes modules et classes. Cela a pour conséquence que *setUpClass* / *setUpModule* (etc) sont appelé exactement une fois par classe et module. Si vous rendez l'ordre aléatoire, de sorte que les tests de différents modules et classes soient adjacents les uns aux autres, alors ces fonctions d'aménagements partagées peuvent être appelées plusieurs fois dans un même test.

Les aménagements de tests partagés ne sont pas conçus pour fonctionner avec des suites dont la commande n'est pas standard. Une `BaseTestSuite` existe toujours pour les cadriciels qui ne veulent pas gérer les aménagements de tests partagés.

S'il y a des exceptions levées pendant l'une des fonctions d'aménagement de tests partagés, le test est signalé comme étant en erreur. Parce qu'il n'y a pas d'instance de test correspondante, un objet `_ErrorHandler` (qui a la même interface qu'une classe `TestCase`) est créé pour représenter l'erreur. Si vous n'utilisez que le lanceur de test unitaire standard, ce détail n'a pas d'importance, mais si vous êtes un auteur de cadriciel de test, il peut être pertinent.

Classes de mise en place (*setUpClass*) et de démantèlement des tests (*tearDownClass*)

Elles doivent être implémentées en tant que méthodes de classe :

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

Si vous voulez que les classes de base `setUpClass` et `tearDownClass` soient appelées, vous devez les appeler vous-même. Les implémentations dans `TestCase` sont vides.

Si une exception est levée pendant l'exécution de `setUpClass` alors les tests dans la classe ne sont pas exécutés et la classe `tearDownClass` n'est pas exécutée. Les classes ignorées n'auront pas d'exécution de `setUpClass` ou `tearDownClass`. Si l'exception est une exception `SkipTest` alors la classe est signalée comme ayant été ignorée au lieu d'être en échec.

Module de mise en place (*setUpModule*) et de démantèlement des tests (*tearDownModule*)

Elles doivent être implémentées en tant que fonctions :

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

Si une exception est levée pendant l'exécution de la fonction `setUpModule` alors aucun des tests du module ne sera exécuté et la fonction `tearDownModule` ne sera pas exécutée. Si l'exception est une exception `SkipTest` alors le module est signalé comme ayant été ignoré au lieu d'être en échec.

26.4.10 Traitement des signaux

Nouveau dans la version 3.2.

L'option `-c/--catch` en ligne de commande pour `unittest`, ainsi que le paramètre `catchbreak` vers `unittest.main()`, permettent une utilisation simplifiée du contrôle-C pendant un test. Avec l'activation de `catchbreak`, l'utilisation du contrôle-C permet de terminer le test en cours d'exécution, et le test se termine et rapporte tous les résultats obtenus jusqu'à présent. Un deuxième contrôle-C lève une exception classique `KeyboardInterrupt`.

Le gestionnaire du signal *contrôle-C* tente de rester compatible avec le code ou les tests qui installent leur propre gestionnaire `signal.SIGINT`. Si le gestionnaire `unittest` est appelé mais *n'est pas* le gestionnaire installé `signal.SIGINT`, c'est-à-dire qu'il a été remplacé par le système sous test et délégué, alors il appelle le gestionnaire par défaut. C'est normalement le comportement attendu par un code qui remplace un gestionnaire installé et lui délègue. Pour les tests individuels qui ont besoin que le signal *contrôle-C* « `unittest` » soit désactivée, le décorateur `removeHandler()` peut être utilisé.

Il existe quelques fonctions de support pour les auteurs de cadriciel afin d'activer la fonctionnalité de gestion des *contrôle-C* dans les cadriciels de test.

`unittest.installHandler()`

Installe le gestionnaire *contrôle-c*. Quand un `signal.SIGINT` est reçu (généralement en réponse à l'utilisateur appuyant sur *contrôle-c*) tous les résultats enregistrés vont appeler la méthode `stop()`.

`unittest.registerResult(result)`

Enregistre un objet `TestResult` pour la gestion du *contrôle-C*. L'enregistrement d'un résultat stocke une référence faible sur celui-ci, de sorte qu'il n'empêche pas que le résultat soit collecté par le ramasse-miette.

L'enregistrement d'un objet `TestResult` n'a pas d'effets de bord si la gestion du *contrôle-c* n'est pas activée, donc les cadriciels de test peuvent enregistrer sans condition tous les résultats qu'ils créent indépendamment du fait que la gestion soit activée ou non.

`unittest.removeResult(result)`

Supprime un résultat enregistré. Une fois qu'un résultat a été supprimé, `stop()` n'est plus appelé sur cet objet résultat en réponse à un *contrôle-c*.

`unittest.removeHandler(function=None)`

Lorsqu'elle est appelée sans arguments, cette fonction supprime le gestionnaire *contrôle-c* s'il a été installé. Cette fonction peut également être utilisée comme décorateur de test pour supprimer temporairement le gestionnaire pendant l'exécution du test :

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

26.5 unittest.mock — Bibliothèque d'objets simulacres

Nouveau dans la version 3.3.

Code source : [Lib/unittest/mock.py](https://github.com/python/cpython/blob/master/Lib/unittest/mock.py)

`unittest.mock` est une bibliothèque pour tester en Python. Elle permet de remplacer des parties du système sous tests par des objets simulacres et faire des assertions sur la façon dont ces objets ont été utilisés.

`unittest.mock` fournit une classe `Mock` pour ne pas avoir besoin de créer manuellement des objets factices dans la suite de tests. Après avoir effectué une action, on peut faire des assertions sur les méthodes / attributs utilisés et les

arguments avec lesquels ils ont été appelés. On peut également spécifier des valeurs renvoyées et définir les attributs nécessaires aux tests.

De plus, *mock* fournit un décorateur `patch()` qui est capable de *patcher* les modules et les classes dans la portée d'un test, ainsi que *sentinel* pour créer des objets uniques. Voir le guide rapide *quick guide* pour quelques exemples d'utilisation de *Mock*, *MagicMock* et `patch()`.

Mock est très facile à utiliser et est conçu pour être utilisé avec *unittest*. *Mock* est basé sur le modèle *action -> assertion* au lieu de *enregistrement -> rejouer* utilisé par de nombreux cadriciels d'objets simulacres.

Il y a un portage de *unittest.mock* pour les versions antérieures de Python, disponible [sur PyPI](#).

26.5.1 Guide rapide

Les classes *Mock* et *MagicMock* créent tous les attributs et méthodes au fur et à mesure des accès et stockent les détails de la façon dont ils ont été utilisés. On peut les configurer, pour spécifier des valeurs de renvoi ou limiter les attributs utilisables, puis faire des assertions sur la façon dont ils ont été utilisés :

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

L'attribut `side_effect` permet de spécifier des effets de bords, y compris la levée d'une exception lorsqu'un objet simulacre est appelé :

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Il existe beaucoup d'autres façons de configurer et de contrôler le comportement de *Mock*. Par exemple, l'argument *spec* configure le *mock* pour qu'il utilise les spécifications d'un autre objet. Tenter d'accéder à des attributs ou méthodes sur le *mock* qui n'existent pas sur l'objet *spec* lève une *AttributeError*.

Le décorateur / gestionnaire de contexte `patch()` permet de simuler facilement des classes ou des objets dans un module sous tests. L'objet spécifié est remplacé par un objet simulacre (ou autre) pendant le test et est restauré à la fin du test :

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
```

(suite sur la page suivante)

(suite de la page précédente)

```

...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()

```

Note : When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `module.ClassName1` is passed in first.

Avec `patch()`, il est important de *patcher* les objets dans l'espace de nommage où ils sont recherchés. C'est ce qui se fait normalement, mais pour un guide rapide, lisez *où patcher*.

Comme tout décorateur, `patch()` peut être utilisé comme gestionnaire de contexte avec une instruction *with* :

```

>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)

```

Il existe également `patch.dict()` pour définir des valeurs d'un dictionnaire au sein d'une portée et restaurer ce dictionnaire à son état d'origine lorsque le test se termine :

```

>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original

```

Mock gère le remplacement des *méthodes magiques* de Python. La façon la plus simple d'utiliser les méthodes magiques est la classe *MagicMock*. Elle permet de faire des choses comme :

```

>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()

```

Mock permet d'assigner des fonctions (ou d'autres instances *Mock*) à des méthodes magiques et elles seront appelées correctement. La classe *MagicMock* est juste une variante de *Mock* qui a toutes les méthodes magiques pré-crées (enfin, toutes les méthodes utiles).

L'exemple suivant est un exemple de création de méthodes magiques avec la classe *Mock* ordinaire :

```

>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='whewheeee')
>>> str(mock)
'whewheeee'

```

Pour être sûr que les objets simulacres dans vos tests ont la même API que les objets qu'ils remplacent, utilisez *l'auto-spécification*. L'auto-spécification peut se faire via l'argument *autospec* de `patch` ou par la fonction

`create_autospec()`. L'auto-spécification crée des objets simulacres qui ont les mêmes attributs et méthodes que les objets qu'ils remplacent, et toutes les fonctions et méthodes (y compris les constructeurs) ont les mêmes signatures d'appel que l'objet réel.

Ceci garantit que vos objets simulacres échouent de la même manière que votre code de production s'ils ne sont pas utilisés correctement :

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
...
TypeError: <lambda>() takes exactly 3 arguments (1 given)
```

La fonction `create_autospec()` peut aussi être utilisée sur les classes, où elle copie la signature de la méthode `__init__`, et sur les objets appelables où elle copie la signature de la méthode `__call__`.

26.5.2 La classe *Mock*

La classe *Mock* est un objet simulacre flexible destiné à remplacer l'utilisation d'objets bouchons et factices dans votre code. Les Mocks sont appelables et créent des attributs comme de nouveaux *Mocks* lorsque l'on y accède¹. L'accès au même attribut renvoie toujours le même *mock*. Les simulacres enregistrent la façon dont ils sont utilisés, ce qui permet de faire des assertions sur ce que le code leur a fait.

La classe *MagicMock* est une sous-classe de *Mock* avec toutes les méthodes magiques pré-crées et prête à l'emploi. Il existe également des variantes non appelables, utiles lorsque l'on simule des objets qui ne sont pas appelables : *NonCallableMock* et *NonCallableMagicMock*.

Le décorateur `patch()` facilite le remplacement temporaire de classes d'un module avec un objet *Mock*. Par défaut `patch()` crée un *MagicMock*. On peut spécifier une classe alternative de *Mock* en utilisant le paramètre `new_callable` de `patch()`.

```
class unittest.mock.Mock(spec=None, side_effect=None, return_value=DEFAULT, wraps=None,
                           name=None, spec_set=None, unsafe=False, **kwargs)
```

Crée un nouvel objet *Mock*. *Mock* prend plusieurs arguments optionnels qui spécifient le comportement de l'objet *Mock* :

- *spec* : une liste de chaînes de caractères ou un objet existant (une classe ou une instance) qui sert de spécification pour l'objet simulacre. Si on passe un objet, alors une liste de chaînes de caractères est formée en appelant la fonction `dir` sur l'objet (à l'exclusion des attributs et méthodes magiques non pris en charge). L'accès à un attribut qui n'est pas dans cette liste entraîne la levée d'une exception *AttributeError*. Si *spec* est un objet (plutôt qu'une liste de chaînes de caractères) alors `__class__` renvoie la classe de l'objet spécifié. Ceci permet aux *mocks* de passer les tests `isinstance()`.
- *spec_set* : variante plus stricte de *spec*. S'il est utilisé, essayer d'utiliser la fonction `set` ou tenter d'accéder à un attribut sur le *mock* qui n'est pas sur l'objet passé comme *spec_set* lève une exception *AttributeError*.
- *side_effect* : fonction à appeler à chaque fois que le *Mock* est appelé. Voir l'attribut `side_effect`. Utile pour lever des exceptions ou modifier dynamiquement les valeurs de retour. La fonction est appelée avec les mêmes

1. The only exceptions are magic methods and attributes (those that have leading and trailing double underscores). Mock doesn't create these but instead raises an *AttributeError*. This is because the interpreter will often implicitly request these methods, and gets very confused to get a new Mock object when it expects a magic method. If you need magic method support see *magic methods*.

arguments que la fonction simulée et, à moins qu'elle ne renvoie `DEFAULT`, la valeur de retour de cette fonction devient la valeur de retour de la fonction simulée.

`side_effect` peut être soit une classe, soit une instance d'exception. Dans ce cas, l'exception est levée lors de l'appel de l'objet simulacre.

Si `side_effect` est un itérable alors chaque appel au `mock` renvoie la valeur suivante de l'itérable.

Utilisez `None` pour remettre à zéro un `side_effect`.

- `return_value` : valeur renvoyée lors de l'appel de l'objet simulacre. Par défaut, il s'agit d'un nouveau `Mock` (créé lors du premier accès). Voir l'attribut `return_value`.

- `unsafe` : par défaut, si un attribut commence par `assert` ou `assert`, une exception `AttributeError` est levée. Le fait de passer `unsafe=True` permet d'accéder à ces attributs.

Nouveau dans la version 3.5.

- `wraps` : élément que le simulacre doit simuler. Si `wraps` n'est pas `None` alors appeler `Mock` passe l'appel à l'objet simulé (renvoyant le résultat réel). L'accès à un attribut sur le `mock` renvoie un objet `Mock` qui simule l'attribut correspondant de l'objet simulé (donc essayer d'accéder à un attribut qui n'existe pas lève une exception `AttributeError`).

Si l'objet simulacre a un ensemble explicite de `return_value` alors les appels ne sont pas passés à l'objet simulé et c'est `return_value` qui est renvoyée à la place.

- `name` : Si le `mock` a un nom, il est alors utilisé par la fonction `repr` du `mock`. C'est utile pour le débogage. Le nom est propagé aux enfants de l'objet `mock`.

Les `mocks` peuvent aussi être appelés avec des arguments par mots-clés arbitraires. Ceux-ci sont utilisés pour définir les attributs sur le `mock` après sa création. Voir la méthode `configure_mock()` pour plus de détails.

assert_called(*args, **kwargs)

Asserter que le `mock` a été appelé au moins une fois.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

Nouveau dans la version 3.6.

assert_called_once(*args, **kwargs)

Asserter que le `mock` a été appelé exactement une fois.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
```

Nouveau dans la version 3.6.

assert_called_with(*args, **kwargs)

Cette méthode est un moyen pratique d'asserter que les appels sont effectués d'une manière particulière :

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(*args, **kwargs)

Asserter que le simulacre a été appelé exactement une fois et que cet appel était avec les arguments spécifiés.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

assert_any_call (*args, **kwargs)

Asserter que le simulacre a été appelé avec les arguments spécifiés.

Asserter que le simulacre a *bien* été appelé avec les arguments au cours de la vie du simulacre. Contrairement à `assert_called_with()` et `assert_called_once_with()` qui passent seulement si l'appel demandé correspond bien au dernier appel, et dans le cas de `assert_called_once_with()` l'appel au simulacre doit être unique.

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls (calls, any_order=False)

Asserter que le simulacre a été appelé avec les appels spécifiés. L'attribut `mock_calls` est comparé à la liste des appels.

Si `any_order` est faux (la valeur par défaut) alors les appels doivent être séquentiels. Il peut y avoir des appels supplémentaires avant ou après les appels spécifiés.

Si `any_order` est vrai alors les appels peuvent être dans n'importe quel ordre, mais ils doivent tous apparaître dans `mock_calls`.

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called ()

Asserter que le simulacre n'a jamais été appelé.

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
```

Nouveau dans la version 3.5.

reset_mock (*, return_value=False, side_effect=False)

La méthode `reset_mock` réinitialise tous les attributs d'appel sur un simulacre :

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
```

(suite sur la page suivante)

(suite de la page précédente)

```
True
>>> mock.reset_mock()
>>> mock.called
False
```

Modifié dans la version 3.6 : Ajout de deux arguments nommés à la fonction `reset_mock`.

Utile pour faire une série d'assertions qui réutilisent le même objet. Attention `reset_mock()` ne réinitialise pas la valeur de retour, les `side_effect` ou tout attribut enfant que vous avez défini en utilisant l'affectation normale par défaut. Pour réinitialiser `return_value` ou `side_effect`, utiliser les paramètres correspondants avec la valeur `True`. Les simulacres enfants et le simulacre de valeur de retour (le cas échéant) seront également réinitialisés.

Note : `return_value`, et `side_effect` sont utilisable uniquement par arguments nommés.

mock_add_spec (*spec*, *spec_set=False*)

Ajoute une spécification à un simulacre. *spec* peut être un objet ou une liste de chaînes de caractères. Seuls les attributs de la spécification *spec* peuvent être récupérés en tant qu'attributs du simulacre.

Si *spec_set* est vrai, seuls les attributs de la spécification peuvent être définis.

attach_mock (*mock*, *attribute*)

Attache un simulacre comme attribut de l'instance courante, en remplaçant son nom et son parent. Les appels au simulacre attaché sont enregistrés dans les attributs `method_calls` et `mock_calls` de l'instance courante.

configure_mock (***kwargs*)

Définir les attributs sur le simulacre à l'aide d'arguments nommés.

Les attributs, les valeurs de retour et les effets de bords peuvent être définis sur des simulacres enfants en utilisant la notation par points standard et en dépaquetant un dictionnaire dans l'appel de méthode :

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
...
KeyError
```

La même chose peut être réalisée en utilisant le constructeur des simulacres :

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
...
KeyError
```

`configure_mock()` existe pour faciliter la configuration après la création du simulacre.

__dir__ ()

Les objets `Mock` limitent les résultats de `dir(un_mock)` à des résultats utiles. Pour les simulacres avec une spécification *spec*, cela inclut tous les attributs autorisés du simulacre.

Voir `FILTER_DIR` pour savoir ce que fait ce filtrage, et comment le désactiver.

_get_child_mock (**kw)

Crée les simulacres enfants pour les attributs et la valeur de retour. Par défaut, les objets simulacre enfants sont du même type que le parent. Les sous-classes de *Mock* peuvent surcharger cette méthode pour personnaliser la façon dont les simulacres enfants sont créés.

Pour les simulacres non appelables, la variante callable est utilisée (plutôt qu'une sous-classe personnalisée).

called

Un booléen représentant si le simulacre a bien été appelé ou non :

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

Un entier indiquant combien de fois le simulacre a été appelé :

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

Définir cette option pour configurer la valeur renvoyé par appel du simulacre :

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

La valeur de revoie par défaut est un simulacre configurable normalement :

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()' id='...'>
>>> mock.return_value.assert_called_with()
```

L'attribut *return_value* peut également être défini dans le constructeur :

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

C'est soit une fonction à appeler lors de l'appel du simulacre, soit une exception (classe ou instance) à lever.

Si vous passez une fonction, elle est appelée avec les mêmes arguments que la fonction simulée et à moins que la fonction ne renvoie le singleton *DEFAULT* l'appel le la fonction simulée renvoie ce que la fonction renvoie.

Si la fonction renvoie *DEFAULT* alors le simulacre renvoie sa valeur normale (celle de *return_value*).

Si vous passez un itérable, il est utilisé pour récupérer un itérateur qui doit renvoyer une valeur à chaque appel. Cette valeur peut être soit une instance d'exception à lever, soit une valeur à renvoyer à l'appel au simulacre (le traitement *DEFAULT* est identique au renvoie de la fonction simulée).

Un exemple d'un simulacre qui lève une exception (pour tester la gestion des exceptions d'une API) :

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
...
Exception: Boom!
```

Utiliser `side_effect` pour renvoyer une séquence de valeurs :

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

Utilisation d'un objet callable :

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

Un attribut `side_effect` peut être défini dans le constructeur. Voici un exemple qui ajoute un à la valeur du simulacre appelé et qui le renvoie :

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

Positionner `side_effect` sur `None` l'efface :

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

C'est soit `None` (si le simulacre n'a pas été appelé), soit les arguments avec lesquels le simulacre a été appelé en dernier. Le retour est sous la forme d'un tuple : le premier élément est l'ensemble des arguments ordonnés avec lequel le simulacre a été appelé (ou un tuple vide) et le second élément est l'ensemble des arguments nommés (ou un dictionnaire vide).

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
```

(suite sur la page suivante)

(suite de la page précédente)

```

>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')

```

L'attribut `call_args`, ainsi que les éléments des listes `call_args_list`, `method_calls` et `mock_calls` sont des objets `call`. Ce sont des tuples, que l'on peut dépaqueter afin de faire des affirmations plus complexes sur chacun des arguments. Voir *appels comme tuples*.

call_args_list

This is a list of all the calls made to the mock object in sequence (so the length of the list is the number of times it has been called). Before any calls have been made it is an empty list. The `call` object can be used for conveniently constructing lists of calls to compare with `call_args_list`.

```

>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True

```

Members of `call_args_list` are `call` objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

method_calls

As well as tracking calls to themselves, mocks also track calls to methods and attributes, and *their* methods and attributes :

```

>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]

```

Members of `method_calls` are `call` objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

mock_calls

`mock_calls` records *all* calls to the mock object, its methods, magic methods *and* return value mocks.

```

>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>

```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

Members of `mock_calls` are `call` objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

Note : The way `mock_calls` are recorded means that where nested calls are made, the parameters of ancestor calls are not recorded and so will always compare equal :

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

`__class__`

Normally the `__class__` attribute of an object will return its type. For a mock object with a spec, `__class__` returns the spec class instead. This allows mock objects to pass `isinstance()` tests for the object they are replacing / masquerading as :

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

`__class__` is assignable to, this allows a mock to pass an `isinstance()` check without forcing you to use a spec :

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class `unittest.mock.NonCallableMock` (*spec=None, wraps=None, name=None, spec_set=None, **kwargs*)

A non-callable version of `Mock`. The constructor parameters have the same meaning of `Mock`, with the exception of `return_value` and `side_effect` which have no meaning on a non-callable mock.

Mock objects that use a class or an instance as a spec or spec_set are able to pass `isinstance()` tests :

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

The `Mock` classes have support for mocking magic methods. See *magic methods* for the full details.

The mock classes and the `patch()` decorators all take arbitrary keyword arguments for configuration. For the `patch()` decorators the keywords are passed to the constructor of the mock being created. The keyword arguments are for configuring attributes of the mock :

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

The return value and side effect of child mocks can be set in the same way, using dotted notation. As you can't use dotted names directly in a call you have to create a dictionary and unpack it using `**` :

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
...
KeyError
```

A callable mock which was created with a *spec* (or a *spec_set*) will introspect the specification object's signature when matching calls to the mock. Therefore, it can match the actual call's arguments regardless of whether they were passed positionally or by name :

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

This applies to `assert_called_with()`, `assert_called_once_with()`, `assert_has_calls()` and `assert_any_call()`. When *Autospeccing*, it will also apply to method calls on the mock object.

Modifié dans la version 3.4 : Added signature introspection on specced and autospecced mock objects.

class `unittest.mock.PropertyMock(*args, **kwargs)`

A mock intended to be used as a property, or other descriptor, on a class. *PropertyMock* provides `__get__()` and `__set__()` methods so you can specify a return value when it is fetched.

Fetching a *PropertyMock* instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> mock_foo.mock_calls
[call(), call(6)]
```

Because of the way mock attributes are stored you can't directly attach a `PropertyMock` to a mock object. Instead you can attach it to the mock type object :

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

Calling

Mock objects are callable. The call will return the value set as the `return_value` attribute. The default return value is a new Mock object; it is created the first time the return value is accessed (either explicitly or by calling the Mock) - but it is stored and the same one returned each time.

Calls made to the object will be recorded in the attributes like `call_args` and `call_args_list`.

If `side_effect` is set then it will be called after the call has been recorded, so if `side_effect` raises an exception the call is still recorded.

The simplest way to make a mock raise an exception when called is to make `side_effect` an exception class or instance :

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

If `side_effect` is a function then whatever that function returns is what calls to the mock return. The `side_effect` function is called with the same arguments as the mock. This allows you to vary the return value of the call dynamically, based on the input :

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

If you want the mock to still return the default return value (a new mock), or any set return value, then there are two ways of doing this. Either return `mock.return_value` from inside `side_effect`, or return `DEFAULT` :

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

To remove a `side_effect`, and return to the default behaviour, set the `side_effect` to `None` :

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

The `side_effect` can also be any iterable object. Repeated calls to the mock will return values from the iterable (until the iterable is exhausted and a `StopIteration` is raised) :

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
...
StopIteration
```

If any members of the iterable are exceptions they will be raised instead of returned :

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
...
ValueError
>>> m()
66
```


Deleting Attributes

Mock objects create attributes on demand. This allows them to pretend to be objects of any type.

You may want a mock object to return `False` to a `hasattr()` call, or raise an `AttributeError` when an attribute is fetched. You can do this by providing an object as a `spec` for a mock, but that isn't always convenient.

You « block » attributes by deleting them. Once deleted, accessing an attribute will raise an `AttributeError`.

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
...
AttributeError: f
```

Mock names and the name attribute

Since « name » is an argument to the `Mock` constructor, if you want your mock object to have a « name » attribute you can't just pass it in at creation time. There are two alternatives. One option is to use `configure_mock()` :

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

A simpler option is to simply set the « name » attribute after mock creation :

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

Attaching Mocks as Attributes

When you attach a mock as an attribute of another mock (or as the return value) it becomes a « child » of that mock. Calls to the child are recorded in the `method_calls` and `mock_calls` attributes of the parent. This is useful for configuring child mocks and then attaching them to the parent, or for attaching mocks to a parent that records all calls to the children and allows you to make assertions about the order of calls between mocks :

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

The exception to this is if the mock has a name. This allows you to prevent the « parenting » if for some reason you don't want it to happen.

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

Mocks created for you by `patch()` are automatically given names. To attach mocks that have names to a parent you use the `attach_mock()` method :

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

26.5.3 The patchers

The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

patch

Note : `patch()` is straightforward to use. The key is to do the patching in the right namespace. See the section [where to patch](#).

`unittest.mock.patch(target, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)`

`patch()` acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the `target` is patched with a `new` object. When the function/with statement exits the patch is undone. If `new` is omitted, then the target is replaced with a `MagicMock`. If `patch()` is used as a decorator and `new` is omitted, the created mock is passed in as an extra argument to the decorated function. If `patch()` is used as a context manager the created mock is returned by the context manager.

`target` should be a string in the form `'package.module.ClassName'`. The `target` is imported and the specified object replaced with the `new` object, so the `target` must be importable from the environment you are calling `patch()` from. The target is imported when the decorated function is executed, not at decoration time.

The `spec` and `spec_set` keyword arguments are passed to the `MagicMock` if patch is creating one for you.

In addition you can pass `spec=True` or `spec_set=True`, which causes patch to pass in the object being mocked as the `spec/spec_set` object.

`new_callable` allows you to specify a different class, or callable object, that will be called to create the `new` object. By default `MagicMock` is used.

A more powerful form of *spec* is *autospec*. If you set `autospec=True` then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a *TypeError* if they are called with the wrong signature. For mocks replacing a class, their return value (the “instance”) will have the same spec as the class. See the `create_autospec()` function and *Autospeccing*.

Instead of `autospec=True` you can pass `autospec=some_object` to use an arbitrary object as the spec instead of the one being replaced.

By default `patch()` will fail to replace attributes that don’t exist. If you pass in `create=True`, and the attribute doesn’t exist, patch will create the attribute for you when the patched function is called, and delete it again afterwards. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don’t actually exist!

Note : Modifié dans la version 3.5 : If you are patching builtins in a module then you don’t need to pass `create=True`, it will be added by default.

Patch can be used as a `TestCase` class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. `patch()` finds tests by looking for method names that start with `patch.TEST_PREFIX`. By default this is `'test'`, which matches the way *unittest* finds tests. You can specify an alternative prefix by setting `patch.TEST_PREFIX`.

Patch can be used as a context manager, with the `with` statement. Here the patching applies to the indented block after the `with` statement. If you use `« as »` then the patched object will be bound to the name after the `« as »`; very useful if `patch()` is creating a mock object for you.

`patch()` takes arbitrary keyword arguments. These will be passed to the *Mock* (or *new_callable*) on construction. `patch.dict(...)`, `patch.multiple(...)` and `patch.object(...)` are available for alternate use-cases.

`patch()` as function decorator, creating the mock for you and passing it into the decorated function :

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

Patching a class replaces the class with a *MagicMock* instance. If the class is instantiated in the code under test then it will be the *return_value* of the mock that will be used.

If the class is instantiated multiple times you could use *side_effect* to return a new mock each time. Alternatively you can set the *return_value* to be anything you want.

To configure return values on methods of *instances* on the patched class you must do this on the *return_value*. For example :

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
... 
```

If you use *spec* or *spec_set* and *patch()* is replacing a *class*, then the return value of the created mock will have the same spec.

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

The *new_callable* argument is useful where you want to use an alternative class to the default *MagicMock* for the created mock. For example, if you wanted a *NonCallableMock* to be used :

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
...
TypeError: 'NonCallableMock' object is not callable
```

Another use case might be to replace an object with an *io.StringIO* instance :

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

When *patch()* is creating a mock for you, it is common that the first thing you need to do is to configure the mock. Some of that configuration can be done in the call to patch. Any arbitrary keywords you pass into the call will be used to set attributes on the created mock :

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

As well as attributes on the created mock attributes, like the *return_value* and *side_effect*, of child mocks can also be configured. These aren't syntactically valid to pass in directly as keyword arguments, but a dictionary with these as keys can still be expanded into a *patch()* call using **** :

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
KeyError
```

patch.object

`patch.object(target, attribute, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)`

patch the named member (*attribute*) on an object (*target*) with a mock object.

`patch.object()` can be used as a decorator, class decorator or a context manager. Arguments *new*, *spec*, *create*, *spec_set*, *autospec* and *new_callable* have the same meaning as for `patch()`. Like `patch()`, `patch.object()` takes arbitrary keyword arguments for configuring the mock object it creates.

When used as a class decorator `patch.object()` honours `patch.TEST_PREFIX` for choosing which methods to wrap.

You can either call `patch.object()` with three arguments or two arguments. The three argument form takes the object to be patched, the attribute name and the object to replace the attribute with.

When calling with the two argument form you omit the replacement object, and a mock is created for you and passed in as an extra argument to the decorated function :

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
... test()
```

spec, *create* and the other arguments to `patch.object()` have the same meaning as they do for `patch()`.

patch.dict

`patch.dict(in_dict, values=(), clear=False, **kwargs)`

Patch a dictionary, or dictionary like object, and restore the dictionary to its original state after the test.

in_dict can be a dictionary or a mapping like container. If it is a mapping then it must at least support getting, setting and deleting items plus iterating over keys.

in_dict can also be a string specifying the name of the dictionary, which will then be fetched by importing it.

values can be a dictionary of values to set in the dictionary. *values* can also be an iterable of (*key*, *value*) pairs.

If *clear* is true then the dictionary will be cleared before the new values are set.

`patch.dict()` can also be called with arbitrary keyword arguments to set values in the dictionary.

`patch.dict()` can be used as a context manager, decorator or class decorator. When used as a class decorator `patch.dict()` honours `patch.TEST_PREFIX` for choosing which methods to wrap.

`patch.dict()` can be used to add members to a dictionary, or simply let a test change a dictionary, and ensure the dictionary is restored when the test ends.

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

Keywords can be used in the `patch.dict()` call to set values in the dictionary :

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

`patch.dict()` can be used with dictionary like objects that aren't actually dictionaries. At the very minimum they must support item getting, setting, deleting and either iteration or membership test. This corresponds to the magic methods `__getitem__()`, `__setitem__()`, `__delitem__()` and either `__iter__()` or `__contains__()`.

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

`patch.multiple(target, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)`

Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches :

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
```

Use `DEFAULT` as the value if you want `patch.multiple()` to create mocks for you. In this case the created mocks are passed into a decorated function by keyword, and a dictionary is returned when `patch.multiple()` is used as a context manager.

`patch.multiple()` can be used as a decorator, class decorator or a context manager. The arguments *spec*, *spec_set*, *create*, *autospec* and *new_callable* have the same meaning as for `patch()`. These arguments will be applied to *all* patches done by `patch.multiple()`.

When used as a class decorator `patch.multiple()` honours `patch.TEST_PREFIX` for choosing which methods to wrap.

If you want `patch.multiple()` to create mocks for you, then you can use `DEFAULT` as the value. If you use `patch.multiple()` as a decorator then the created mocks are passed into the decorated function by keyword.

```
>>> thing = object()
>>> other = object()
```

```
>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

`patch.multiple()` can be nested with other patch decorators, but put arguments passed by keyword *after* any of the standard arguments created by `patch()` :

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

If `patch.multiple()` is used as a context manager, the value returned by the context manager is a dictionary where created mocks are keyed by name :

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
>>>
```

patch methods : start and stop

All the patchers have `start()` and `stop()` methods. These make it simpler to do patching in `setUp` methods or where you want to do multiple patches without nesting decorators or with statements.

To use them call `patch()`, `patch.object()` or `patch.dict()` as normal and keep a reference to the returned patcher object. You can then call `start()` to put the patch in place and `stop()` to undo it.

If you are using `patch()` to create a mock for you then it will be returned by the call to `patcher.start`.

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

A typical use case for this might be for doing multiple patches in the `setUp` method of a `TestCase` :

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

Prudence : If you use this technique you must ensure that the patching is « undone » by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the `setUp` then `tearDown` is not called. `unittest.TestCase.addCleanup()` makes this easier :

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
... 
```

As an added bonus you no longer need to keep a reference to the patcher object.

It is also possible to stop all patches which have been started by using `patch.stopall()`.

`patch.stopall()`

Stop all active patches. Only stops patches started with `start`.

patch builtins

You can patch any builtins within a module. The following example patches builtin `ord()` :

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

All of the patchers can be used as class decorators. When used in this way they wrap every test method on the class. The patchers recognise methods that start with 'test' as being test methods. This is the same way that the `unittest.TestLoader` finds test methods by default.

It is possible that you want to use a different prefix for your tests. You can inform the patchers of the different prefix by setting `patch.TEST_PREFIX` :

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

Nesting Patch Decorators

If you want to perform multiple patches then you can simply stack up the decorators.

You can stack up multiple patch decorators using this pattern :

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

Note that the decorators are applied from the bottom upwards. This is the standard way that Python applies decorators. The order of the created mocks passed into your test function matches this order.

Where to patch

`patch()` works by (temporarily) changing the object that a *name* points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

The basic principle is that you patch where an object is *looked up*, which is not necessarily the same place as where it is defined. A couple of examples will help to clarify this.

Imagine we have a project that we want to test with the following structure :

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

Now we want to test `some_function` but we want to mock out `SomeClass` using `patch()`. The problem is that when we import module `b`, which we will have to do then it imports `SomeClass` from module `a`. If we use `patch()` to mock out `a.SomeClass` then it will have no effect on our test ; module `b` already has a reference to the *real* `SomeClass` and it looks like our patching had no effect.

The key is to patch out `SomeClass` where it is used (or where it is looked up). In this case `some_function` will actually look up `SomeClass` in module `b`, where we have imported it. The patching should look like :

```
@patch('b.SomeClass')
```

However, consider the alternative scenario where instead of `from a import SomeClass` module `b` does `import a` and `some_function` uses `a.SomeClass`. Both of these import forms are common. In this case the class we want to patch is being looked up in the module and so we have to patch `a.SomeClass` instead :

```
@patch('a.SomeClass')
```

Patching Descriptors and Proxy Objects

Both `patch` and `patch.object` correctly patch and restore descriptors : class methods, static methods and properties. You should patch these on the *class* rather than an instance. They also work with *some* objects that proxy attribute access, like the `django settings` object.

26.5.4 MagicMock and magic method support

Mocking Magic Methods

Mock supports mocking the Python protocol methods, also known as « magic methods ». This allows mock objects to replace containers or other objects that implement Python protocols.

Because magic methods are looked up differently from normal methods², this support has been specially implemented. This means that only specific magic methods are supported. The supported list includes *almost* all of them. If there are any missing that you need please let us know.

You mock magic methods by setting the method you are interested in to a function or a mock instance. If you are using a function then it *must* take `self` as the first argument³.

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

One use case for this is for mocking objects used as context managers in a `with` statement :

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Calls to magic methods do not appear in *method_calls*, but they are recorded in *mock_calls*.

Note : If you use the *spec* keyword argument to create a mock then attempting to set a magic method that isn't in the spec will raise an *AttributeError*.

The full list of supported magic methods is :

- `__hash__`, `__sizeof__`, `__repr__` and `__str__`
- `__dir__`, `__format__` et `__subclasses__`

2. Magic methods *should* be looked up on the class rather than the instance. Different versions of Python are inconsistent about applying this rule. The supported protocol methods should work with all supported versions of Python.

3. The function is basically hooked up to the class, but each `Mock` instance is kept isolated from the others.

- `__floor__`, `__trunc__` et `__ceil__`
- Comparisons : `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` and `__ne__`
- Container methods : `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__reversed__` and `__missing__`
- Context manager : `__enter__` and `__exit__`
- Unary numeric methods : `__neg__`, `__pos__` and `__invert__`
- The numeric methods (including right hand and in-place variants) : `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__div__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__`, and `__pow__`
- Numeric conversion methods : `__complex__`, `__int__`, `__float__` and `__index__`
- Descriptor methods : `__get__`, `__set__` and `__delete__`
- Pickling : `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`

The following methods exist but are *not* supported as they are either in use by mock, can't be set dynamically, or can cause problems :

- `__getattr__`, `__setattr__`, `__init__` and `__new__`
- `__prepare__`, `__instancecheck__`, `__subclasscheck__`, `__del__`

Magic Mock

There are two `MagicMock` variants : `MagicMock` and `NonCallableMagicMock`.

class `unittest.mock.MagicMock(*args, **kw)`

`MagicMock` is a subclass of `Mock` with default implementations of most of the magic methods. You can use `MagicMock` without having to configure the magic methods yourself.

The constructor parameters have the same meaning as for `Mock`.

If you use the `spec` or `spec_set` arguments then *only* magic methods that exist in the spec will be created.

class `unittest.mock.NonCallableMagicMock(*args, **kw)`

A non-callable version of `MagicMock`.

The constructor parameters have the same meaning as for `MagicMock`, with the exception of `return_value` and `side_effect` which have no meaning on a non-callable mock.

The magic methods are setup with `MagicMock` objects, so you can configure them and use them in the usual way :

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

By default many of the protocol methods are required to return objects of a specific type. These methods are preconfigured with a default return value, so that they can be used without you having to do anything if you aren't interested in the return value. You can still *set* the return value manually if you want to change the default.

Methods and their defaults :

- `__lt__` : `NotImplemented`
- `__gt__` : `NotImplemented`
- `__le__` : `NotImplemented`
- `__ge__` : `NotImplemented`
- `__int__` : `1`
- `__contains__` : `False`
- `__len__` : `0`
- `__iter__` : `iter([])`

```

— __exit__: False
— __complex__: 1j
— __float__: 1.0
— __bool__: True
— __index__: 1
— __hash__: default hash for the mock
— __str__: default str for the mock
— __sizeof__: default sizeof for the mock

```

Par exemple :

```

>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False

```

The two equality methods, `__eq__()` and `__ne__()`, are special. They do the default equality comparison on identity, using the `side_effect` attribute, unless you change their return value to return something else :

```

>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True

```

The return value of `MagicMock.__iter__()` can be any iterable object and isn't required to be an iterator :

```

>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']

```

If the return value is an iterator, then iterating over it once will consume it and subsequent iterations will result in an empty list :

```

>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]

```

`MagicMock` has all of the supported magic methods configured except for some of the obscure and obsolete ones. You can still set these up if you want.

Magic methods that are supported but not setup by default in `MagicMock` are :

```

— __subclasses__
— __dir__
— __format__
— __get__, __set__ et __delete__

```

```
— __reversed__ et __missing__  
— __reduce__, __reduce_ex__, __getinitargs__, __getnewargs__, __getstate__ and  
  __setstate__  
— __getformat__ et __setformat__
```

26.5.5 Helpers

sentinel

`unittest.mock.sentinel`

The `sentinel` object provides a convenient way of providing unique objects for your tests.

Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object. The objects returned have a sensible repr so that test failure messages are readable.

The `sentinel` attributes don't preserve their identity when they are *copied* or *pickled*.

Sometimes when testing you need to test that a specific object is passed as an argument to another method, or returned. It can be common to create named sentinel objects to test this. `sentinel` provides a convenient way of creating and testing the identity of objects like this.

In this example we monkey patch method to return `sentinel.some_object` :

```
>>> real = ProductionClass()  
>>> real.method = Mock(name="method")  
>>> real.method.return_value = sentinel.some_object  
>>> result = real.method()  
>>> assert result is sentinel.some_object  
>>> sentinel.some_object  
sentinel.some_object
```

DEFAULT

`unittest.mock.DEFAULT`

The `DEFAULT` object is a pre-created sentinel (actually `sentinel.DEFAULT`). It can be used by *side_effect* functions to indicate that the normal return value should be used.

call

`unittest.mock.call(*args, **kwargs)`

`call()` is a helper object for making simpler assertions, for comparing with `call_args`, `call_args_list`, `mock_calls` and `method_calls`. `call()` can also be used with `assert_has_calls()`.

```
>>> m = MagicMock(return_value=None)  
>>> m(1, 2, a='foo', b='bar')  
>>> m()  
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]  
True
```

`call.call_list()`

For a call object that represents multiple calls, `call_list()` returns a list of all the intermediate calls as well as the final call.

`call_list` is particularly useful for making assertions on « chained calls ». A chained call is multiple calls on a single line of code. This results in multiple entries in `mock_calls` on a mock. Manually constructing the sequence of calls can be tedious.

`call_list()` can construct the sequence of calls from the same chained call :

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1,
  call().method(arg='foo'),
  call().method().other('bar'),
  call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

A call object is either a tuple of (positional args, keyword args) or (name, positional args, keyword args) depending on how it was constructed. When you construct them yourself this isn't particularly interesting, but the `call` objects that are in the `Mock.call_args`, `Mock.call_args_list` and `Mock.mock_calls` attributes can be introspected to get at the individual arguments they contain.

The call objects in `Mock.call_args` and `Mock.call_args_list` are two-tuples of (positional args, keyword args) whereas the call objects in `Mock.mock_calls`, along with ones you construct yourself, are three-tuples of (name, positional args, keyword args).

You can use their « tupleness » to pull out the individual arguments for more complex introspection and assertions. The positional arguments are a tuple (an empty tuple if there are no positional arguments) and the keyword arguments are a dictionary :

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> args, kwargs = kall
>>> args
(1, 2, 3)
>>> kwargs
{'arg2': 'two', 'arg': 'one'}
>>> args is kall[0]
True
>>> kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg2': 'three', 'arg': 'two'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

`unittest.mock.create_autospec(spec, spec_set=False, instance=False, **kwargs)`

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the *spec* object as their spec.

Functions or methods being mocked will have their arguments checked to ensure that they are called with the correct signature.

If *spec_set* is `True` then attempting to set attributes that don't exist on the spec object will raise an `AttributeError`.

If a class is used as a spec then the return value of the mock (the instance of the class) will have the same spec. You can use a class as the spec for an instance object by passing *instance=True*. The returned mock will only be callable if instances of the mock are callable.

`create_autospec()` also takes arbitrary keyword arguments that are passed to the constructor of the created mock.

See *Autospeccing* for examples of how to use auto-speccing with `create_autospec()` and the *autospec* argument to `patch()`.

ANY

`unittest.mock.ANY`

Sometimes you may need to make assertions about *some* of the arguments in a call to mock, but either not care about some of the arguments or want to pull them individually out of *call_args* and make more complex assertions on them.

To ignore certain arguments you can pass in objects that compare equal to *everything*. Calls to `assert_called_with()` and `assert_called_once_with()` will then succeed no matter what was passed in.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

ANY can also be used in comparisons with call lists like *mock_calls*:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

FILTER_DIR

`unittest.mock.FILTER_DIR`

FILTER_DIR is a module level variable that controls the way mock objects respond to `dir()` (only for Python 2.6 or more recent). The default is `True`, which uses the filtering described below, to only show useful members. If you dislike this filtering, or need to switch it off for diagnostic purposes, then set `mock.FILTER_DIR = False`.

With filtering on, `dir(some_mock)` shows only useful attributes and will include any dynamically created attributes that wouldn't normally be shown. If the mock was created with a *spec* (or *autospec* of course) then all the attributes from the original are shown, even if they haven't been accessed yet:


```
>>> dir(Mock())
['assert_any_call',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...]
```

Many of the not-very-useful (private to *Mock* rather than the thing being mocked) underscore and double underscore prefixed attributes have been filtered from the result of calling *dir()* on a *Mock*. If you dislike this behaviour you can switch it off by setting the module level switch *FILTER_DIR*:

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...]
```

Alternatively you can just use *vars(my_mock)* (instance members) and *dir(type(my_mock))* (type members) to bypass the filtering irrespective of *mock.FILTER_DIR*.

mock_open

`unittest.mock.mock_open(mock=None, read_data=None)`

A helper function to create a mock to replace the use of *open()*. It works for *open()* called directly or used as a context manager.

The *mock* argument is the mock object to configure. If *None* (the default) then a *MagicMock* will be created for you, with the API limited to methods or attributes available on standard file handles.

read_data is a string for the *read()*, *readline()*, and *readlines()* methods of the file handle to return. Calls to those methods will take data from *read_data* until it is depleted. The mock of these methods is pretty simplistic : every time the *mock* is called, the *read_data* is rewound to the start. If you need more control over the data that you are feeding to the tested code you will need to customize this mock for yourself. When that is insufficient, one of the in-memory filesystem packages on PyPI can offer a realistic filesystem for testing.

Modifié dans la version 3.4 : Added *readline()* and *readlines()* support. The mock of *read()* changed to consume *read_data* rather than returning it on each call.

Modifié dans la version 3.5 : *read_data* is now reset on each call to the *mock*.

Using *open()* as a context manager is a great way to ensure your file handles are closed properly and is becoming common :

```
with open('/some/path', 'w') as f:
    f.write('something')
```

The issue is that even if you mock out the call to `open()` it is the *returned object* that is used as a context manager (and has `__enter__()` and `__exit__()` called).

Mocking context managers with a *MagicMock* is common enough and fiddly enough that a helper function is useful.

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

And for reading files :

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

Autospeccing

Autospeccing is based on the existing *spec* feature of *mock*. It limits the api of mocks to the api of an original object (the *spec*), but it is recursive (implemented lazily) so that attributes of mocks only have the same api as the attributes of the *spec*. In addition mocked functions / methods have the same call signature as the original so they raise a *TypeError* if they are called incorrectly.

Before I explain how auto-speccing works, here's why it is needed.

Mock is a very powerful and flexible object, but it suffers from two flaws when used to mock out objects from a system under test. One of these flaws is specific to the *Mock* api and the other is a more general problem with using mock objects.

First the problem specific to *Mock*. *Mock* has two assert methods that are extremely handy : *assert_called_with()* and *assert_called_once_with()*.

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
Traceback (most recent call last):
...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

Because mocks auto-create attributes on demand, and allow you to call them with arbitrary arguments, if you misspell one of these assert methods then your assertion is gone :

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assret_called_once_with(4, 5, 6)
```

Your tests can pass silently and incorrectly because of the typo.

The second issue is more general to mocking. If you refactor some of your code, rename members and so on, any tests for code that is still using the *old api* but uses mocks instead of the real objects will still pass. This means your tests can all pass even though your code is broken.

Note that this is another reason why you need integration tests as well as unit tests. Testing everything in isolation is all fine and dandy, but if you don't test how your units are « wired together » there is still lots of room for bugs that tests might have caught.

mock already provides a feature to help with this, called speccing. If you use a class or instance as the `spec` for a mock then you can only access attributes on the mock that exist on the real class :

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assert_called_with
Traceback (most recent call last):
...
AttributeError: Mock object has no attribute 'assert_called_with'
```

The spec only applies to the mock itself, so we still have the same issue with any methods on the mock :

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assert_called_with()
```

Auto-speccing solves this problem. You can either pass `autospec=True` to `patch()` / `patch.object()` or use the `create_autospec()` function to create a mock with a spec. If you use the `autospec=True` argument to `patch()` then the object that is being replaced will be used as the spec object. Because the speccing is done « lazily » (the spec is created as attributes on the mock are accessed) you can use it with very complex or deeply nested objects (like modules that import modules that import modules) without a big performance hit.

Here's an example of it in use :

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='... '>
```

You can see that `request.Request` has a spec. `request.Request` takes two arguments in the constructor (one of which is *self*). Here's what happens if we try to call it incorrectly :

```
>>> req = request.Request()
Traceback (most recent call last):
...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

The spec also applies to instantiated classes (i.e. the return value of specced mocks) :

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='... '>
```

`Request` objects are not callable, so the return value of instantiating our mocked out `request.Request` is a non-callable mock. With the spec in place any typos in our asserts will raise the correct error :

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='... '>
>>> req.add_header.assert_called_with
Traceback (most recent call last):
...
AttributeError: Mock object has no attribute 'assert_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

In many cases you will just be able to add `autospec=True` to your existing `patch()` calls and then be protected against bugs due to typos and api changes.

As well as using *autospec* through `patch()` there is a `create_autospec()` for creating autospecced mocks directly :

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='... '>
```

This isn't without caveats and limitations however, which is why it is not the default behaviour. In order to know what attributes are available on the spec object, *autospec* has to introspect (access attributes) the spec. As you traverse attributes on the mock a corresponding traversal of the original object is happening under the hood. If any of your specced objects have properties or descriptors that can trigger code execution then you may not be able to use *autospec*. On the other hand it is much better to design your objects so that introspection is safe⁴.

A more serious problem is that it is common for instance attributes to be created in the `__init__()` method and not to exist on the class at all. *autospec* can't know about any dynamically created attributes and restricts the api to visible attributes.

```
>>> class Something:
...     def __init__(self):
...         self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...     thing = Something()
...     thing.a
...
Traceback (most recent call last):
...
AttributeError: Mock object has no attribute 'a'
```

There are a few different ways of resolving this problem. The easiest, but not necessarily the least annoying, way is to simply set the required attributes on the mock after creation. Just because *autospec* doesn't allow you to fetch attributes that don't exist on the spec it doesn't prevent you setting them :

```
>>> with patch('__main__.Something', autospec=True):
...     thing = Something()
...     thing.a = 33
...
...

```

There is a more aggressive version of both *spec* and *autospec* that *does* prevent you setting non-existent attributes. This is useful if you want to ensure your code only *sets* valid attributes too, but obviously it prevents this particular scenario :

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...     thing = Something()
```

(suite sur la page suivante)

4. This only applies to classes or already instantiated objects. Calling a mocked class to create a mock instance *does not* create a real instance. It is only attribute lookups - along with calls to `dir()` - that are done.

(suite de la page précédente)

```
...     thing.a = 33
...
Traceback (most recent call last):
...
AttributeError: Mock object has no attribute 'a'
```

Probably the best way of solving the problem is to add class attributes as default values for instance members initialised in `__init__()`. Note that if you are only setting default attributes in `__init__()` then providing them via class attributes (shared between instances of course) is faster too. e.g.

```
class Something:
    a = 33
```

This brings up another issue. It is relatively common to provide a default value of `None` for members that will later be an object of a different type. `None` would be useless as a spec because it wouldn't let you access *any* attributes or methods on it. As `None` is *never* going to be useful as a spec, and probably indicates a member that will normally of some other type, `autospec` doesn't use a spec for members that are set to `None`. These will just be ordinary mocks (well - `MagicMocks`) :

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

If modifying your production classes to add defaults isn't to your liking then there are more options. One of these is simply to use an instance as the spec rather than the class. The other is to create a subclass of the production class and add the defaults to the subclass without affecting the production class. Both of these require you to use an alternative object as the spec. Thankfully `patch()` supports this - you can simply pass the alternative object as the *autospec* argument :

```
>>> class Something:
...     def __init__(self):
...         self.a = 33
...
>>> class SomethingForTest(Something):
...     a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

26.6 unittest.mock — getting started

Nouveau dans la version 3.3.

26.6.1 Utilisation de Mock ou l'art de singer

Simulation des méthodes

Usages courant de *Mock* :

- Substitution des méthodes
- Enregistrement des appels faits sur les objets

On peut remplacer une méthode sur un objet pour contrôler qu'elle est bien appelée avec le nombre correct d'arguments :

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

Une fois notre objet simulacre appelé (via `real.method` dans notre exemple), il fournit des méthodes et attributs permettant de valider comment il a été appelé.

Note : Dans la majeure partie des exemples donnés ici, les classes *Mock* et *MagicMock* sont interchangeables. Étant donné que *MagicMock* est la classe la plus puissante des deux, cela fait sens de l'utiliser par défaut.

Une fois l'objet *Mock* appelé, son attribut `called` est défini à `True`. Qui plus est, nous pouvons utiliser les méthodes `assert_called_with()` ou `assert_called_once_with()` pour contrôler qu'il a été appelé avec les bons arguments.

Cet exemple teste que l'appel de la méthode `ProductionClass().method` implique bien celui de la méthode `something` :

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

S'assurer de la bonne utilisation d'un objet

Dans l'exemple précédent, nous avons directement remplacé une méthode par un objet (afin de valider que l'appel était correct). Une autre façon de faire est de passer un objet *Mock* en argument d'une méthode (ou de tout autre partie du code à tester) et ensuite de contrôler que notre objet a été utilisé de la façon attendue.

Ci-dessous, `ProductionClass` dispose d'une méthode `closer`. Si on l'appelle avec un objet, alors elle appelle la méthode `close` dessus.

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
... 
```

Ainsi, pour tester cette classe, nous devons lui passer un objet ayant une méthode `close`, puis vérifier qu'elle a bien été appelée.

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

En fait, nous n'avons pas à nous soucier de fournir la méthode `close` dans notre objet « simulé ». Le simple fait d'accéder à la méthode `close` l'a créée. Si par contre la méthode `close` n'a pas été appelée alors, bien que le test la crée en y accédant, `assert_called_with()` lèvera une exception.

Simulation des classes

Un cas d'utilisation courant consiste à émuler les classes instanciées par le code que nous testons. Quand on *patch* une classe, alors cette classe est remplacée par un objet *mock*. Les instances de la classe étant créées en *appelant la classe*, on accède à « l'instance *mock* » via la valeur de retour de la classe émulée.

In the example below we have a function `some_function` that instantiates `Foo` and calls a method on it. The call to `patch()` replaces the class `Foo` with a mock. The `Foo` instance is the result of calling the mock, so it is configured by modifying the mock `return_value`.

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

Naming your mocks

It can be useful to give your mocks a name. The name is shown in the repr of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock :

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

Tracking all Calls

Often you want to track more than a single call to a method. The `mock_calls` attribute records all calls to child attributes of the mock - and also to their children.

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

If you make an assertion about `mock_calls` and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls :

You use the `call` object to construct lists for comparing with `mock_calls` :

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

However, parameters to calls that return mocks are not recorded, which means it is not possible to track nested calls where the parameters used to create ancestors are important :

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy :

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock :

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

The return value can also be set in the constructor :

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

If you need an attribute setting on your mock, just do it :


```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

Sometimes you want to mock up a more complex situation, like for example `mock.connection.cursor().execute("SELECT 1")`. If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use `call` to construct the set of calls in a « chained call » like this for easy assertion afterwards :

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

It is the call to `.call_list()` that turns our call object into a list of calls representing the chained calls.

Raising exceptions with mocks

A useful attribute is `side_effect`. If you set this to an exception class or instance then the exception will be raised when the mock is called.

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
...
Exception: Boom!
```

Side effect functions and iterables

`side_effect` can also be set to a function or an iterable. The use case for `side_effect` as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set `side_effect` to an iterable every call to the mock returns the next value from the iterable :

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, `side_effect` can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns :

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that *also* provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` - then your tests will continue to pass even though your code is now broken !

`Mock` allows you to provide an object as a specification for the mock, using the `spec` keyword argument. Accessing methods / attributes on the mock that don't exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
...
AttributeError: object has no attribute 'old_method'
```

Using a specification also enables a smarter matching of calls made to the mock, regardless of whether some parameters were passed as positional or named arguments :

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

If you want this smarter matching to also work with method calls on the mock, you can use *auto-specing*.

If you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use `spec_set` instead of `spec`.

26.6.2 Patch Decorators

Note : Avec `patch()`, il est important de *patcher* les objets dans l'espace de nommage où ils sont recherchés. C'est ce qui se fait normalement, mais pour un guide rapide, lisez *où patcher*.

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

mock provides three convenient decorators for this : `patch()`, `patch.object()` and `patch.dict()`. `patch` takes a single string, of the form `package.module.Class.attribute` to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. “`patch.object`” takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

`patch.object` :

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original
```

```
>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

If you are patching a module (including *builtins*) then use `patch()` instead of `patch.object()` :

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

The module name can be “dotted”, in the form `package.module` if needed :

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

A nice pattern is to actually decorate test methods themselves :

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

If you want to patch with a Mock, you can use `patch()` with only one argument (or `patch.object()` with two arguments). The mock will be created for you and passed into the test function / method :

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
```

(suite sur la page suivante)

(suite de la page précédente)

```

...     SomeClass.static_method()
...     mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()

```

You can stack up multiple patch decorators using this pattern :

```

>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()

```

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `test_module.ClassName2` is passed in first.

Il existe également `patch.dict()` pour définir des valeurs d'un dictionnaire au sein d'une portée et restaurer ce dictionnaire à son état d'origine lorsque le test se termine :

```

>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original

```

`patch`, `patch.object` and `patch.dict` can all be used as context managers.

Where you use `patch()` to create a mock for you, you can get a reference to the mock using the « as » form of the with statement :

```

>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)

```

As an alternative `patch`, `patch.object` and `patch.dict` can be used as class decorators. When used in this way it is the same as applying the decorator individually to every method whose name starts with « test ».

26.6.3 Further Examples

Here are some more examples for some slightly more advanced scenarios.

Mocking chained calls

Mocking chained calls is actually straightforward with `mock` once you understand the `return_value` attribute. When a mock is called for the first time, or you fetch its `return_value` before it has been called, a new `Mock` is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the `return_value` mock :

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place...

So, suppose we have some code that looks a little bit like this :

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'eggs
↳').start_call()
...         # more code
```

Assuming that `BackendProvider` is already well tested, how do we test `method()` ? Specifically, we want to test that the code section `# more code` uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the `backend` attribute on a `Something` instance. In this particular case we are only interested in the return value from the final call to `start_call` so we don't have much configuration to do. Let's assume the object it returns is "file-like", so we'll ensure that our response object uses the builtin `open()` as its spec.

To do this we create a mock instance as our mock backend and create a mock response object for it. To set the response as the return value for that final `start_call` we could do this :

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_
↳value = mock_response
```

We can do that in a slightly nicer way using the `configure_mock()` method to directly set the return value for us :

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.return_
↳value': mock_response}
>>> mock_backend.configure_mock(**config)
```

With these we monkey patch the « mock backend » in place and can make the real call :

```
>>> something.backend = mock_backend
>>> something.method()
```

Using `mock_calls` we can check the chained call with a single assert. A chained call is several calls in one line of code, so there will be several entries in `mock_calls`. We can use `call.call_list()` to create this list of calls for us :

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Partial mocking

In some tests I wanted to mock out a call to `datetime.date.today()` to return a known date, but I didn't want to prevent the code under test from creating new date objects. Unfortunately `datetime.date` is written in C, and so I couldn't just monkey-patch out the static `date.today()` method.

I found a simple way of doing this that involved effectively wrapping the date class with a mock, but passing through calls to the constructor to the real class (and returning real instances).

The `patch decorator` is used here to mock out the date class in the module under test. The `side_effect` attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class is called a real date will be constructed and returned by `side_effect`.

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
... 
```

Note that we don't patch `datetime.date` globally, we patch `date` in the module that *uses* it. See [where to patch](#).

When `date.today()` is called a known date is returned, but calls to the `date(...)` constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

Calls to the date constructor are recorded in the `mock_date` attributes (`call_count` and `friends`) which may also be useful for your tests.

An alternative way of dealing with mocking dates, or other builtin classes, is discussed in [this blog entry](#).

Mocking a Generator Method

A Python generator is a function or method that uses the `yield` statement to return a series of values when iterated over¹.

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is `__iter__()`, so we can mock this using a `MagicMock`.

Here's an example class with an « iter » method implemented as a generator :

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
```

(suite sur la page suivante)

1. There are also generator expressions and more [advanced uses](#) of generators, but we aren't concerned about them here. A very good introduction to generators and how powerful they are is : [Generator Tricks for Systems Programmers](#).

(suite de la page précédente)

```
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

How would we mock this class, and in particular its « iter » method ?

To configure the values returned from the iteration (implicit in the call to `list`), we need to configure the object returned by the call to `foo.iter()`.

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. For Python 2.6 or more recent you can use `patch()` (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test` :

```
>>> @patch('mymodule.SomeClass')
... class MyTest(TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

An alternative way of managing patches is to use the *patch methods : start and stop*. These allow you to move the patching into your `setUp` and `tearDown` methods.

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

If you use this technique you must ensure that the patching is « undone » by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the `setUp` then `tearDown` is not called. `unittest.TestCase.addCleanup()` makes this easier :

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

Mocking Unbound Methods

Whilst writing tests today I needed to patch an *unbound method* (patching the method on the class rather than on the instance). I needed `self` to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can't patch with a mock for this, because if you replace an unbound method with a mock it doesn't become a bound method when fetched from the instance, and so it doesn't get `self` passed in. The workaround is to patch the unbound method with a real function instead. The `patch()` decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `autospec=True` to patch then it does the patching with a *real* function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have `self` passed in as the first argument, which is exactly what I wanted :

```
>>> class Foo:
...     def foo(self):
...         pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...     mock_foo.return_value = 'foo'
...     foo = Foo()
...     foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

If we don't use `autospec=True` then the unbound method is patched out with a `Mock` instance instead, and isn't called with `self`.

Checking multiple calls with mock

mock has a nice API for making assertions about how your mock objects are used.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

If your mock is only being called once you can use the `assert_called_once_with()` method that also asserts that the `call_count` is one.

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
...
AssertionError: Expected to be called once. Called 2 times.
```

Both `assert_called_with` and `assert_called_once_with` make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls you can use `call_args_list`:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

The `call` helper makes it easy to make assertions about these calls. You can build up a list of expected calls and compare it to `call_args_list`. This looks remarkably similar to the repr of the `call_args_list`:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

Coping with mutable arguments

Another situation is rare, but can bite you, is when your mock is called with mutable arguments. `call_args` and `call_args_list` store *references* to the arguments. If the arguments are mutated by the code under test then you can no longer make assertions about what the values were when the mock was called.

Here's some example code that shows the problem. Imagine the following functions defined in “mymodule”:

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that `grob` calls `frob` with the correct argument look what happens :

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.

Here's one solution that uses the `side_effect` functionality. If you provide a `side_effect` function for a mock then `side_effect` will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using *another* mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

`copy_call_args` is called with the mock that will be called. It returns a new mock that we do the assertion on. The `side_effect` function makes a copy of the args and calls our `new_mock` with the copy.

Note : If your mock is only going to be used once there is an easier way of checking arguments at the point they are called. You can simply do the checking inside a `side_effect` function.

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
...
AssertionError
```

An alternative approach is to create a subclass of `Mock` or `MagicMock` that copies (using `copy.deepcopy()`) the arguments. Here's an example implementation :

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super(CopyingMock, self).__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
...
AssertionError: Expected call: mock({1})
Actual call: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

When you subclass `Mock` or `MagicMock` all dynamically created attributes, and the `return_value` will use your subclass automatically. That means all children of a `CopyingMock` will also have the type `CopyingMock`.

Nesting Patches

Using `patch` as a context manager is nice, but if you do multiple patches you can end up with nested `with` statements indenting further and further to the right :

```
>>> class MyTest(TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

With unittest cleanup functions and the *patch methods : start and stop* we can achieve the same effect without the nested indentation. A simple helper method, `create_patch`, puts the patch in place and returns the created mock for us :

```
>>> class MyTest(TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
... 
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

Mocking a dictionary with MagicMock

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with *MagicMock*, which will behave like a dictionary, and using *side_effect* to delegate dictionary access to a real underlying dictionary that is under our control.

When the `__getitem__()` and `__setitem__()` methods of our *MagicMock* are called (normal dictionary access) then *side_effect* is called with the key (and in the case of `__setitem__` the value too). We can also control what is returned.

After the *MagicMock* has been used we can use attributes like *call_args_list* to assert about how the dictionary was used :

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...     return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

Note : An alternative to using *MagicMock* is to use *Mock* and *only* provide the magic methods you specifically want :

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

A *third* option is to use *MagicMock* but passing in `dict` as the *spec* (or *spec_set*) argument so that the *MagicMock* created only has dictionary magic methods available :

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the `mock` will behave like a normal dictionary but recording the access. It even raises a *KeyError* if you try to access a key that doesn't exist.

```

>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'

```

After it has been used you can make assertions about the access using the normal mock methods and attributes :

```

>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'c': 3, 'b': 'fish', 'd': 'eggs'}

```

Mock subclasses and their attributes

There are various reasons why you might want to subclass `Mock`. One reason might be to add helper methods. Here's a silly example :

```

>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True

```

The standard behaviour for `Mock` instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that `Mock` attributes are `Mocks` and `MagicMock` attributes are `MagicMocks`². So if you're subclassing to add helper methods then they'll also be available on the attributes and return value mock of instances of your subclass.

```

>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>

```

(suite sur la page suivante)

2. An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn't have callable methods.

(suite de la page précédente)

```
>>> mymock.foo.has_been_called()
True
```

Sometimes this is inconvenient. For example, [one user](#) is subclassing mock to create a [Twisted adaptor](#). Having this applied to attributes too actually causes errors.

Mock (in all its flavours) uses a method called `_get_child_mock` to create these « sub-mocks » for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (`**kwargs`) which are then passed onto the mock constructor :

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

Mocking imports with `patch.dict`

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren't using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is *usually* a much better way to solve the problem (refactor the code) or to prevent « up front costs » by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use mock to affect the results of an import. Importing fetches an *object* from the `sys.modules` dictionary. Note that it fetches an *object*, which need not be a module. Importing a module for the first time results in a module object being put in `sys.modules`, so usually when you import something you get a module back. This need not be the case however.

This means you can use `patch.dict()` to temporarily put a mock in place in `sys.modules`. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the “fooble” module.

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the import `fooble` succeeds, but on exit there is no “fooble” left in `sys.modules`.

This also works for the `from module import name` form :

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='... '>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports :

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='... '>
>>> mock.module.fooble.assert_called_once_with()
```

Tracking order of calls and less verbose call assertions

The *Mock* class allows you to track the *order* of method calls on your mock objects through the *method_calls* attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use *mock_calls* to achieve the same effect.

Because mocks track calls to child mocks in *mock_calls*, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the *mock_calls* of the parent :

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='... '>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='... '>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

We can then assert about the calls, including the order, by comparing with the *mock_calls* attribute on the manager mock :

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

If patch is creating, and putting in place, your mocks then you can attach them to a manager mock using the *attach_mock()* method. After attaching calls will be recorded in *mock_calls* of the manager.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
```

(suite sur la page suivante)

(suite de la page précédente)

```

...     manager.attach_mock(MockClass2, 'MockClass2')
...     MockClass1().foo()
...     MockClass2().bar()
...
<MagicMock name='mock.MockClass1().foo()' id='... '>
<MagicMock name='mock.MockClass2().bar()' id='... '>
>>> manager.mock_calls
[call.MockClass1(),
 call.MockClass1().foo(),
 call.MockClass2(),
 call.MockClass2().bar()]

```

If many calls have been made, but you're only interested in a particular sequence of them then an alternative is to use the `assert_has_calls()` method. This takes a list of calls (constructed with the `call` object). If that sequence of calls are in `mock_calls` then the assert succeeds.

```

>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='... '>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='... '>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)

```

Even though the chained call `m.one().two().three()` aren't the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about *some* of those calls. You may not even care about the order. In this case you can pass `any_order=True` to `assert_has_calls`:

```

>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)

```

More complex argument matching

Using the same basic concept as `ANY` we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use `assert_called_with()` we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a “standard” call to `assert_called_with` isn't sufficient:

```

>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))

```

(suite sur la page suivante)

(suite de la page précédente)

```
Traceback (most recent call last):
...
AssertionError: Expected: call(<__main__.Foo object at 0x...>)
Actual call: call(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this :

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this :

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
...
```

Putting all this together :

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The `Matcher` is instantiated with our `compare` function and the `Foo` object we want to compare against. In `assert_called_with` the `Matcher` equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don't an `AssertionError` is raised :

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

With a bit of tweaking you could have the comparison function raise the `AssertionError` directly and provide a more useful failure message.

As of version 1.5, the Python testing library `PyHamcrest` provides similar functionality, that may be useful here, in the form of its equality matcher (`hamcrest.library.integration.match_equality`).

26.7 2to3 — Traduction automatique de code en Python 2 vers Python 3

2to3 est un programme Python qui lit du code source en Python 2.x et applique une suite de correcteurs pour le transformer en code Python 3.x valide. La bibliothèque standard contient un ensemble riche de correcteurs qui gèreront quasiment tout le code. La bibliothèque *lib2to3* utilisée par *2to3* est cependant une bibliothèque flexible et générique, il est donc possible d'écrire vos propres correcteurs pour *2to3*. *lib2to3* pourrait aussi être adaptée à des applications personnalisées dans lesquelles le code Python doit être édité automatiquement.

26.7.1 Utilisation de 2to3

2to3 sera généralement installé avec l'interpréteur Python en tant que script. Il est également situé dans le dossier `Tools/scripts` à racine de Python.

Les arguments de base de *2to3* sont une liste de fichiers et de répertoires à transformer. Les répertoires sont parcourus récursivement pour trouver les sources Python.

Voici un exemple de fichier source Python 2.x, `example.py` :

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

Il peut être converti en code Python 3.x par *2to3* en ligne de commande :

```
$ 2to3 example.py
```

Une comparaison avec le fichier source original est affichée. *2to3* peut aussi écrire les modifications nécessaires directement dans le fichier source. (Une sauvegarde du fichier d'origine est effectuée à moins que l'option `-n` soit également donnée.) L'écriture des modifications est activée avec l'option `-w` :

```
$ 2to3 -w example.py
```

Après transformation, `example.py` ressemble à :

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

Les commentaires et les retraits sont préservés tout au long du processus de traduction.

Par défaut, *2to3* exécute un ensemble de *correcteurs prédéfinis*. L'option `-l` énumère tous les correcteurs disponibles. Un ensemble explicite de correcteurs à exécuter peut être donné avec `-f`. De même, `-x` désactive explicitement un correcteur. L'exemple suivant exécute uniquement les `import` et les correcteurs `has_key` :

```
$ 2to3 -f imports -f has_key example.py
```

Cette commande exécute tous les correcteurs, sauf le correcteur `apply` :

```
$ 2to3 -x apply example.py
```

Certains correcteurs sont *explicites*, ce qui signifie qu'ils ne sont pas exécutés par défaut et doivent être énumérés sur la ligne de commande à exécuter. Ici, en plus des correcteurs par défaut, le correcteur `idioms` est exécuté :

```
$ 2to3 -f all -f idioms example.py
```

Notez que passer `all` active tous les correcteurs par défaut.

Parfois, `2to3` trouvera un endroit dans votre code source qui doit être changé, mais qu'il ne peut pas résoudre automatiquement. Dans ce cas, `2to3` affiche un avertissement sous la comparaison d'un fichier. Vous devez traiter l'avertissement afin d'avoir un code conforme à Python 3.x.

`2to3` peut également réusiner les *doctests*. Pour activer ce mode, utilisez `-d`. Notez que *seul* les *doctests* seront réusins. Cela ne nécessite pas que le module soit du Python valide. Par exemple, des *doctests* tels que des exemples dans un document *reST* peuvent également être réusins avec cette option.

L'option `-v` augmente la quantité de messages générés par le processus de traduction.

Puisque certaines instructions d'affichage peuvent être analysées comme des appels ou des instructions de fonction, `2to3` ne peut pas toujours lire les fichiers contenant la fonction d'affichage. Lorsque `2to3` détecte la présence de la directive compilateur `from __future__ import print_function`, il modifie sa grammaire interne pour interpréter `print()` comme une fonction. Cette modification peut également être activée manuellement avec l'option `-p`. Utilisez `-p` pour exécuter des correcteurs sur du code dont les instructions d'affichage ont déjà été converties.

L'option `-o` ou `--output-dir` permet de donner autre répertoire pour les fichiers de sortie en écriture. L'option `-n` est requise quand on les utilise comme fichiers de sauvegarde qui n'ont pas de sens si les fichiers d'entrée ne sont pas écrasés.

Nouveau dans la version 3.2.3 : L'option `-o` a été ajoutée.

L'option `-W` ou `--write-unchanged-files` indique à `2to3` de toujours écrire des fichiers de sortie même si aucun changement du fichier n'était nécessaire. Ceci est très utile avec `!-o` pour qu'un arbre des sources Python entier soit copié avec la traduction d'un répertoire à l'autre. Cette option implique `-w` sans quoi elle n'aurait pas de sens.

Nouveau dans la version 3.2.3 : L'option `-W` a été ajoutée.

L'option `--add-suffix` spécifie une chaîne à ajouter à tous les noms de fichiers de sortie. L'option `-n` est nécessaire dans ce cas, puisque sauvegarder n'est pas nécessaire en écrivant dans des fichiers différents. Exemple :

```
$ 2to3 -n -W --add-suffix=3 example.py
```

Écrit un fichier converti nommé `example.py3`.

Nouveau dans la version 3.2.3 : L'option `--add-suffix` est ajoutée.

Pour traduire un projet entier d'une arborescence de répertoires à une autre, utilisez :

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

26.7.2 Correcteurs

Chaque étape de la transformation du code est encapsulée dans un correcteur. La commande `2to3 -l` les énumère. Comme *documenté ci-dessus*, chacun peut être activé ou désactivé individuellement. Ils sont décrits plus en détails ici.

apply

Supprime l'usage d'`apply()`. Par exemple, `apply(function, *args, **kwargs)` est converti en `function(*args, **kwargs)`.

asserts

Remplace les noms de méthodes obsolètes du module `unittest` par les bons.

De	À
<code>failUnlessEqual(a, b)</code>	<code>assertEqual(a, b)</code>
<code>assertEquals(a, b)</code>	<code>assertEqual(a, b)</code>
<code>failIfEqual(a, b)</code>	<code>assertNotEqual(a, b)</code>
<code>assertNotEquals(a, b)</code>	<code>assertNotEqual(a, b)</code>
<code>failUnless(a)</code>	<code>assertTrue(a)</code>
<code>assert_(a)</code>	<code>assertTrue(a)</code>
<code>failIf(a)</code>	<code>assertFalse(a)</code>
<code>failUnlessRaises(exc, cal)</code>	<code>assertRaises(exc, cal)</code>
<code>failUnlessAlmostEqual(a, b)</code>	<code>assertAlmostEqual(a, b)</code>
<code>assertAlmostEquals(a, b)</code>	<code>assertAlmostEqual(a, b)</code>
<code>failIfAlmostEqual(a, b)</code>	<code>assertNotAlmostEqual(a, b)</code>
<code>assertNotAlmostEquals(a, b)</code>	<code>assertNotAlmostEqual(a, b)</code>

basestring

Convertit une `basestring` en `str`.

buffer

Convertit un `buffer` en `memoryview`. Ce correcteur est optionnel car l'API `memoryview` est similaire mais pas exactement pareil que celle de `buffer`.

dict

Fixe les méthodes d'itération sur les dictionnaires. `dict.iteritems()` est converti en `dict.items()`, `dict.iterkeys()` en `dict.keys()` et `dict.itervalues()` en `dict.values()`. De la même façon, `dict.viewitems()`, `dict.viewkeys()` et `dict.viewvalues()` sont convertis respectivement en `dict.items()`, `dict.keys()` et `dict.values()`. Il encapsule également les usages existants de `dict.items()`, `dict.keys()` et `dict.values()` dans un appel à `list`.

except

Convertit `except X, T` en `except X as T`.

exec

Convertit l'instruction `exec` en fonction `exec()`.

execfile

Supprime l'usage de `execfile()`. L'argument de `execfile()` est encapsulé dans des appels à `open()`, `compile()` et `exec()`.

exitfunc

Change l'affectation de `sys.exitfunc` pour utiliser le module `atexit`.

filter

Encapsule l'usage de `filter()` dans un appel à `list`.

funcattrs

Fixe les attributs de fonction ayant été renommés. Par exemple, `my_function.func_closure` est converti en `my_function.__closure__`.

future

Supprime les instructions `from __future__ import new_feature`.

getcwdu

Renomme `os.getcwdu()` en `os.getcwd()`.

has_key

Change `dict.has_key(key)` en `key in dict`.

idioms

Ce correcteur optionnel effectue plusieurs transformations rendant le code Python plus idiomatique. Les com-

paraisons de types telles que `type(x) is SomeClass` et `type(x) == SomeClass` sont converties en `isinstance(x, SomeClass)`. `while 1` devient `while True`. Ce correcteur essaye aussi d'utiliser `sorted()` aux endroits appropriés. Par exemple, ce bloc

```
L = list(some_iterable)
L.sort()
```

est transformé en

```
L = sorted(some_iterable)
```

import

Détecte les importations voisines et les convertit en importations relatives.

imports

Gère les renommages de modules dans la bibliothèque standard.

imports2

Gères d'autres renommages de modules dans la bibliothèque standard. Il est distinct de `imports` seulement en raison de limitations techniques.

input

Convertit `input(prompt)` en `eval(input(prompt))`.

intern

Convertit `intern()` en `sys.intern()`.

isinstance

Fixe les types dupliqués dans le second argument de `isinstance()`. Par exemple, `isinstance(x, (int, int))` est converti en `isinstance(x, int)` et `isinstance(x, (int, float, int))` est converti en `isinstance(x, (int, float))`.

itertools_imports

Supprime les importations de `itertools.ifilter()`, `itertools.izip()` et `itertools.imap()`. Les importations de `itertools.ifilterfalse()` sont aussi changées en `itertools.filterfalse()`.

itertools

Change l'usage de `itertools.ifilter()`, `itertools.izip()` et `itertools.imap()` en leurs équivalents intégrés. `itertools.ifilterfalse()` est changé en `itertools.filterfalse()`.

long

Renomme `long` en `int`.

map

Encapsule `map()` dans un appel à `list`. Change aussi `map(None, x)` en `list(x)`. L'usage de `from future_builtins import map` désactive ce correcteur.

metaclass

Convertit l'ancienne syntaxe de métaclasse (`__metaclass__ = Meta` dans le corps de la classe) à la nouvelle (`class X(metaclass=Meta)`).

methodattrs

Fixe les anciens noms d'attributs de méthodes. Par exemple, `meth.im_func` est converti en `meth.__func__`.

ne

Convertit l'ancienne syntaxe d'inégalité, `<>`, en `!=`.

next

Convertit l'usage des méthodes `next()` de l'itérateur en `next()`. Renomme également les méthodes `next()` en `__next__()`.

nonzero

Renomme `__nonzero__()` en `__bool__()`.

numliterals

Convertit les nombres écrits littéralement en octal dans leur nouvelle syntaxe.

operator

Converts calls to various functions in the `operator` module to other, but equivalent, function calls. When needed, the appropriate import statements are added, e.g. `import collections`. The following mapping are made :

De	A
<code>operator.isCallable(obj)</code>	<code>hasattr(obj, '__call__')</code>
<code>operator.sequenceIncludes(obj)</code>	<code>operator.contains(obj)</code>
<code>operator.isSequenceType(obj)</code>	<code>isinstance(obj, collections.Sequence)</code>
<code>operator.isMappingType(obj)</code>	<code>isinstance(obj, collections.Mapping)</code>
<code>operator.isNumberType(obj)</code>	<code>isinstance(obj, numbers.Number)</code>
<code>operator.repeat(obj, n)</code>	<code>operator.mul(obj, n)</code>
<code>operator.irepeat(obj, n)</code>	<code>operator.imul(obj, n)</code>

paren

Ajoute des parenthèses supplémentaires lorsqu'elles sont nécessaires dans les listes en compréhension. Par exemple, `[x for x in 1, 2]` devient `[x for x in (1, 2)]`.

print

Convertit l'instruction `print` en fonction `print()`.

raise

Convertit `raise E, Ven raise E(V) et raise E, V, Ten raise E(V).with_traceback(T)`. Si `E` est un tuple, la conversion sera incorrecte puisque la substitution de tuples aux exceptions a été supprimée en 3.0.

raw_input

Convertit `raw_input()` en `input()`.

reduce

Gère le déplacement de `reduce()` à `functools.reduce()`.

reload

Convertit les appels à `reload()` en appels à `imp.reload()`.

renames

Change `sys.maxint` en `sys.maxsize`.

repr

Remplace les accents graves utilisés comme `repr` par des appels à `repr()`.

set_literal

Remplace l'usage du constructeur de `set` par les ensembles littéraux. Ce correcteur est optionnel.

standarderror

Renomme `StandardError` en `Exception`.

sys_exc

Change les `sys.exc_value`, `sys.exc_type`, `sys.exc_traceback` dépréciés en `sys.exc_info()`.

throw

Fixe le changement de l'API dans la méthode `throw()` du générateur.

tuple_params

Supprime la décompression implicite des paramètres d'un tuple. Ce correcteur ajoute des variables temporaires.

types

Fixe le code cassé par la suppression de certains membres du module `types`.

unicode

Renomme `unicode` en `str`.

urllib

Gère le renommage des paquets `urllib` et `urllib2` en `urllib`.

ws_comma

Supprime l'espace excédentaire des éléments séparés par des virgules. Ce correcteur est optionnel.

xrange

Renomme la fonction `xrange()` en `range()` et encapsule les appels à la fonction `range()` avec des appels à `list`.

xreadlines

Change `for x in file.xreadlines()` en `for x in file`.

zip

Encapsule l'usage de `zip()` dans un appel à `list`. Ceci est désactivé lorsque `from future_builtins import zip` apparaît.

26.7.3 lib2to3 — la bibliothèque de 2to3

Code source : [Lib/lib2to3/](#)

Note : L'API de `lib2to3` devrait être considérée instable et peut changer drastiquement dans le futur.

26.8 test — Regression tests package for Python

Note : The `test` package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python's standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The `test` package contains all regression tests for Python as well as the modules `test.support` and `test.regrtest`. `test.support` is used to enhance your tests while `test.regrtest` drives the testing suite.

Each module in the `test` package whose name starts with `test_` is a testing suite for a specific module or feature. All new tests should be written using the `unittest` or `doctest` module. Some older tests are written using a « traditional » testing style that compares output printed to `sys.stdout`; this style of test is considered deprecated.

Voir aussi :

Module `unittest` Writing PyUnit regression tests.

Module `doctest` Tests embedded in documentation strings.

26.8.1 Writing Unit Tests for the `test` package

It is preferred that tests that use the `unittest` module follow a few guidelines. One is to name the test module by starting it with `test_` and end it with the name of the module being tested. The test methods in the test module should start with `test_` and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as `# Tests function returns only True or False`) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used :

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

This code pattern allows the testing suite to be run by `test.regrtest`, on its own as a script that supports the `unittest` CLI, or via the `python -m unittest` CLI.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed :

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also « private » code.
- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.
- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.
- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.
- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.

- Make sure to clean up after your tests (such as close and remove all temporary files).
- If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.
- Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.
- Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input :

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

When using this pattern, remember that all classes that inherit from `unittest.TestCase` are run as tests. The `Mixin` class in the example above does not have any data and so can't be run by itself, thus it does not inherit from `unittest.TestCase`.

Voir aussi :

Test Driven Development A book by Kent Beck on writing tests before code.

26.8.2 Running tests using the command-line interface

The `test` package can be run as a script to drive Python's regression test suite, thanks to the `-m` option : **python -m test**. Under the hood, it uses `test.regrtest`; the call **python -m test.regrtest** used in previous Python versions still works. Running the script by itself automatically starts running all regression tests in the `test` package. It does this by finding all modules in the package whose name starts with `test_`, importing them, and executing the function `test_main()` if present or loading the tests via `unittest.TestLoader.loadTestsFromModule` if `test_main` does not exist. The names of tests to execute may also be passed to the script. Specifying a single regression test (**python -m test test_spam**) will minimize output and only print whether the test passed or failed.

Running `test` directly allows what resources are available for tests to use to be set. You do this by using the `-u` command-line option. Specifying `all` as the value for the `-u` option enables all possible resources : **python -m test -uall**. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after `all`. The command **python -m test -uall,-audio,-largefile** will run `test` with all resources except the `audio` and `largefile` resources. For a list of all resources and more command-line options, run **python -m test -h**.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run **make test** at the top-level directory where Python was built. On Windows, executing **rt.bat** from your `PCBuild` directory will run all regression tests.

26.9 test.support — Utilities for the Python test suite

The `test.support` module provides support for Python's regression test suite.

Note : `test.support` is not a public module. It is documented here to help Python developers write tests. The API of this module is subject to change without backwards compatibility concerns between releases.

This module defines the following exceptions :

exception `test.support.TestFailed`

Exception to be raised when a test fails. This is deprecated in favor of `unittest`-based tests and `unittest.TestCase`'s assertion methods.

exception `test.support.ResourceDenied`

Subclass of `unittest.SkipTest`. Raised when a resource (such as a network connection) is not available. Raised by the `requires()` function.

The `test.support` module defines the following constants :

`test.support.verbose`

True when verbose output is enabled. Should be checked when more detailed information is desired about a running test. `verbose` is set by `test.regrtest`.

`test.support.is_jython`

True if the running interpreter is Jython.

`test.support.TESTFN`

Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

The `test.support` module defines the following functions :

`test.support.forget(module_name)`

Remove the module named `module_name` from `sys.modules` and delete any byte-compiled files of the module.

`test.support.is_resource_enabled(resource)`

Return True if `resource` is enabled and available. The list of available resources is only set when `test.regrtest` is executing the tests.

`test.support.requires(resource, msg=None)`

Raise `ResourceDenied` if `resource` is not available. `msg` is the argument to `ResourceDenied` if it is raised. Always returns True if called by a function whose `__name__` is `'__main__'`. Used when tests are executed by `test.regrtest`.

`test.support.findfile(filename, subdir=None)`

Return the path to the file named `filename`. If no match is found `filename` is returned. This does not equal a failure since it could be the path to the file.

Setting `subdir` indicates a relative path to use to find the file rather than looking directly in the path directories.

`test.support.run_unittest(*classes)`

Execute `unittest.TestCase` subclasses passed to the function. The function scans the classes for methods starting with the prefix `test_` and executes the tests individually.

It is also legal to pass strings as parameters; these should be keys in `sys.modules`. Each associated module will be scanned by `unittest.TestLoader.loadTestsFromModule()`. This is usually seen in the following `test_main()` function :

```
def test_main():
    support.run_unittest(__name__)
```

This will run all tests defined in the named module.

`test.support.run_doctest(module, verbosity=None)`

Run `doctest.testmod()` on the given *module*. Return (failure_count, test_count).

If *verbosity* is None, `doctest.testmod()` is run with verbosity set to *verbose*. Otherwise, it is run with verbosity set to None.

`test.support.check_warnings(*filters, quiet=True)`

A convenience wrapper for `warnings.catch_warnings()` that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling `warnings.catch_warnings(record=True)` with `warnings.simplefilter()` set to always and with the option to automatically validate the results that are recorded.

`check_warnings` accepts 2-tuples of the form ("message regexp", `WarningCategory`) as positional arguments. If one or more *filters* are provided, or if the optional keyword argument *quiet* is False, it checks to make sure the warnings are as expected : each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set *quiet* to True.

If no arguments are specified, it defaults to :

```
check_warnings(("", Warning), quiet=True)
```

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a `WarningRecorder` instance is returned. The underlying warnings list from `catch_warnings()` is available via the recorder object's *warnings* attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return None.

The recorder object also has a `reset()` method, which clears the warnings list.

The context manager is designed to be used like this :

```
with check_warnings(("assertion is always true", SyntaxWarning),
                   ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

In this case if either warning was not raised, or some other warning was raised, `check_warnings()` would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used :

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

Modifié dans la version 3.2 : New optional arguments *filters* and *quiet*.

`test.support.captured_stdin()`

`test.support.captured_stdout()`

`test.support.captured_stderr()`

A context managers that temporarily replaces the named stream with *io.StringIO* object.

Example use with output streams :

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

Example use with input stream :

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

`test.support.temp_dir(path=None, quiet=False)`

A context manager that creates a temporary directory at *path* and yields the directory.

If *path* is *None*, the temporary directory is created using *tempfile.mkdtemp()*. If *quiet* is *False*, the context manager raises an exception on error. Otherwise, if *path* is specified and cannot be created, only a warning is issued.

`test.support.change_cwd(path, quiet=False)`

A context manager that temporarily changes the current working directory to *path* and yields the directory.

If *quiet* is *False*, the context manager raises an exception on error. Otherwise, it issues only a warning and keeps the current working directory the same.

`test.support.temp_cwd(name='tempcwd', quiet=False)`

A context manager that temporarily creates a new directory and changes the current working directory (CWD).

The context manager creates a temporary directory in the current directory with name *name* before temporarily changing the current working directory. If *name* is *None*, the temporary directory is created using *tempfile.mkdtemp()*.

If *quiet* is *False* and it is not possible to create or change the CWD, an error is raised. Otherwise, only a warning is raised and the original CWD is used.

`test.support.temp_umask(umask)`

A context manager that temporarily sets the process umask.

`test.support.can_symlink()`

Return *True* if the OS supports symbolic links, *False* otherwise.

`@test.support.skip_unless_symlink`

A decorator for running tests that require support for symbolic links.

`@test.support.anticipate_failure(condition)`

A decorator to conditionally mark tests with *unittest.expectedFailure()*. Any use of this decorator should have an associated comment identifying the relevant tracker issue.

`@test.support.run_with_locale(catstr, *locales)`

A decorator for running a function in a different locale, correctly resetting it after it has finished. *catstr* is the locale category as a string (for example "LC_ALL"). The *locales* passed will be tried sequentially, and the first valid locale will be used.

`test.support.make_bad_fd()`

Create an invalid file descriptor by opening and closing a temporary file, and returning its descriptor.

`test.support.import_module(name, deprecated=False)`

This function imports and returns the named module. Unlike a normal import, this function raises `unittest.SkipTest` if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if `deprecated` is `True`.

Nouveau dans la version 3.1.

`test.support.import_fresh_module(name, fresh=(), blocked=(), deprecated=False)`

This function imports and returns a fresh copy of the named Python module by removing the named module from `sys.modules` before doing the import. Note that unlike `reload()`, the original module is not affected by this operation.

`fresh` is an iterable of additional module names that are also removed from the `sys.modules` cache before doing the import.

`blocked` is an iterable of module names that are replaced with `None` in the module cache during the import to ensure that attempts to import them raise `ImportError`.

The named module and any modules named in the `fresh` and `blocked` parameters are saved before starting the import and then reinserted into `sys.modules` when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if `deprecated` is `True`.

This function will raise `ImportError` if the named module cannot be imported.

Example use :

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

Nouveau dans la version 3.1.

`test.support.bind_port(sock, host=HOST)`

Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the `sock.family` is `AF_INET` and `sock.type` is `SOCK_STREAM`, and the socket has `SO_REUSEADDR` or `SO_REUSEPORT` set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the `SO_EXCLUSIVEADDRUSE` socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.

`test.support.find_unused_port(family=socket.AF_INET, socktype=socket.SOCK_STREAM)`

Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the `sock` parameter (default is `AF_INET`, `SOCK_STREAM`), and binding it to the specified host address (defaults to `0.0.0.0`) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or `bind_port()` should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the `-accept` argument to openssl's `s_server` mode). Always prefer `bind_port()` over `find_unused_port()` where possible. Using a hard coded port is discouraged since it can make multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

`test.support.load_package_tests(pkg_dir, loader, standard_tests, pattern)`

Generic implementation of the `unittest` `load_tests` protocol for use in test packages. `pkg_dir` is the root directory of the package; `loader`, `standard_tests`, and `pattern` are the arguments expected by `load_tests`. In simple cases, the test package's `__init__.py` can be the following :

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

`test.support.detect_api_mismatch(ref_api, other_api, *, ignore=())`

Returns the set of attributes, functions or methods of *ref_api* not found on *other_api*, except for a defined list of items to be ignored in this check specified in *ignore*.

By default this skips private attributes beginning with “_” but includes all magic methods, i.e. those starting and ending in “__”.

Nouveau dans la version 3.5.

`test.support.check__all__(test_case, module, name_of_module=None, extra=(), blacklist=())`

Assert that the `__all__` variable of *module* contains all public names.

The module’s public names (its API) are detected automatically based on whether they match the public name convention and were defined in *module*.

The *name_of_module* argument can specify (as a string or tuple thereof) what module(s) an API could be defined in in order to be detected as a public API. One case for this is when *module* imports part of its public API from other modules, possibly a C backend (like `csv` and its `_csv`).

The *extra* argument can be a set of names that wouldn’t otherwise be automatically detected as « public », like objects without a proper `__module__` attribute. If provided, it will be added to the automatically detected ones. The *blacklist* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.

Example use :

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        blacklist = {'baz'} # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, blacklist=blacklist)
```

Nouveau dans la version 3.6.

The `test.support` module defines the following classes :

class `test.support.TransientResource` (*exc*, ***kwargs*)

Instances are a context manager that raises `ResourceDenied` if the specified exception type is raised. Any keyword arguments are treated as attribute/value pairs to be compared against any exception raised within the `with` statement. Only if all pairs match properly against attributes on the exception is `ResourceDenied` raised.

class `test.support.EnvironmentVarGuard`

Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying `os.environ`. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

Modifié dans la version 3.1 : Added dictionary interface.

`EnvironmentVarGuard.set(envvar, value)`

Temporarily set the environment variable `envvar` to the value of `value`.

`EnvironmentVarGuard.unset(envvar)`

Temporarily unset the environment variable `envvar`.

class `test.support.SuppressCrashReport`

A context manager used to try to prevent crash dialog popups on tests that are expected to crash a subprocess.

On Windows, it disables Windows Error Reporting dialogs using `SetErrorMode`.

On UNIX, `resource.setrlimit()` is used to set `resource.RLIMIT_CORE`'s soft limit to 0 to prevent coredump file creation.

On both platforms, the old value is restored by `__exit__()`.

class `test.support.WarningsRecorder`

Class used to record warnings for unit tests. See documentation of `check_warnings()` above for more details.

class `test.support.FakePath(path)`

Simple *path-like object*. It implements the `__fspath__()` method which just returns the `path` argument. If `path` is an exception, it will be raised in `__fspath__()`.

Débogueur et instrumentation

Ces bibliothèques sont là pour vous aider lors du développement en Python : Le débogueur vous permet d'avancer pas à pas dans le code, d'analyser la pile d'appel, de placer des points d'arrêts, ... Les outils d'instrumentation exécutent du code et vous donnent un rapport détaillé du temps d'exécution, vous permettant d'identifier les goulots d'étranglement dans vos programmes.

27.1 `bdb` — Framework de débogage

Code source : [Lib/bdb.py](#)

The `bdb` module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

L'exception suivante est définie :

exception `bdb.BdbQuit`

Exception raised by the `Bdb` class for quitting the debugger.

Le module `bdb` définit deux classes :

class `bdb.Breakpoint` (*self, file, line, temporary=0, cond=None, funcname=None*)

This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

Breakpoints are indexed by number through a list called `bppbynumber` and by (`file`, `line`) pairs through `bplist`. The former points to a single instance of class `Breakpoint`. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated filename should be in canonical form. If a `funcname` is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

`Breakpoint` instances have the following methods :

deleteMe ()

Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

enable()

Active le point d'arrêt.

disable()

Désactive le point d'arrêt.

bpformat()

Return a string with all the information about the breakpoint, nicely formatted :

- Le numéro du point d'arrêt.
 - S'il est temporaire ou non.
 - Its file, line position.
 - The condition that causes a break.
 - If it must be ignored the next N times.
 - The breakpoint hit count.
- Nouveau dans la version 3.2.

bpprint (*out=None*)

Print the output of *bpformat()* to the file *out*, or if it is *None*, to standard output.

class `bdb.Bdb` (*skip=None*)

The *Bdb* class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (*pdb.Pdb*) is an example.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

Nouveau dans la version 3.1 : L'argument *skip*.

The following methods of *Bdb* normally don't need to be overridden.

canonic (*filename*)

Auxiliary method for getting a filename in a canonical form, that is, as a case-normalized (on case-insensitive filesystems) absolute path, stripped of surrounding angle brackets.

reset()

Set the `botframe`, `stopframe`, `returnframe` and `quitting` attributes with values ready to start debugging.

trace_dispatch (*frame, event, arg*)

This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following :

- "line" : A new line of code is going to be executed.
- "call" : A function is about to be called, or another code block entered.
- "return" : A function or other code block is about to return.
- "exception" : Une exception est survenue.
- "c_call" : Une fonction C est sur le point d'être appelée.
- "c_return" : Une fonction C s'est terminée.
- "c_exception" : A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

Le paramètre *arg* dépend de l'événement précédent.

See the documentation for *sys.settrace()* for more information on the trace function. For more information on code and frame objects, refer to types.

dispatch_line (*frame*)

If the debugger should stop on the current line, invoke the *user_line()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the *Bdb.quitting* flag is set (which can be set from *user_line()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_call (*frame*, *arg*)

If the debugger should stop on this function call, invoke the `user_call()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_call()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

dispatch_return (*frame*, *arg*)

If the debugger should stop on this function return, invoke the `user_return()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_return()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

dispatch_exception (*frame*, *arg*)

If the debugger should stop at this exception, invokes the `user_exception()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_exception()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

Normally derived classes don't override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

stop_here (*frame*)

This method checks if the *frame* is somewhere below `botframe` in the call stack. `botframe` is the frame in which debugging started.

break_here (*frame*)

This method checks if there is a breakpoint in the filename and line belonging to *frame* or, at least, in the current function. If the breakpoint is a temporary one, this method deletes it.

break_anywhere (*frame*)

This method checks if there is a breakpoint in the filename of the current frame.

Derived classes should override these methods to gain control over debugger operation.

user_call (*frame*, *argument_list*)

This method is called from `dispatch_call()` when there is the possibility that a break might be necessary anywhere inside the called function.

user_line (*frame*)

This method is called from `dispatch_line()` when either `stop_here()` or `break_here()` yields `True`.

user_return (*frame*, *return_value*)

This method is called from `dispatch_return()` when `stop_here()` yields `True`.

user_exception (*frame*, *exc_info*)

This method is called from `dispatch_exception()` when `stop_here()` yields `True`.

do_clear (*arg*)

Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step ()

Arrête après une ligne de code.

set_next (*frame*)

Stop on the next line in or below the given frame.

set_return (*frame*)

Stop when returning from the given frame.

set_until (*frame*)

Stop when the line with the line no greater than the current one is reached or when returning from current frame.

set_trace ([*frame*])

Start debugging from *frame*. If *frame* is not specified, debugging starts from caller's frame.

set_continue ()

Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to `None`.

set_quit()

Set the quitting attribute to True. This raises `BdbQuit` in the next call to one of the `dispatch_*()` methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or None if all is well.

set_break(filename, lineno, temporary=0, cond, funcname)

Set a new breakpoint. If the `lineno` line doesn't exist for the `filename` passed as argument, return an error message. The `filename` should be in canonical form, as described in the `canonic()` method.

clear_break(filename, lineno)

Delete the breakpoints in `filename` and `lineno`. If none were set, an error message is returned.

clear_bpbynumber(arg)

Delete the breakpoint which has the index `arg` in the `Breakpoint.bpbynumber`. If `arg` is not numeric or out of range, return an error message.

clear_all_file_breaks(filename)

Delete all breakpoints in `filename`. If none were set, an error message is returned.

clear_all_breaks()

Supprime tous les points d'arrêt définis.

get_bpbynumber(arg)

Return a breakpoint specified by the given number. If `arg` is a string, it will be converted to a number. If `arg` is a non-numeric string, if the given breakpoint never existed or has been deleted, a `ValueError` is raised.

Nouveau dans la version 3.2.

get_break(filename, lineno)

Check if there is a breakpoint for `lineno` of `filename`.

get_breaks(filename, lineno)

Return all breakpoints for `lineno` in `filename`, or an empty list if none are set.

get_file_breaks(filename)

Return all breakpoints in `filename`, or an empty list if none are set.

get_all_breaks()

Donne tous les points d'arrêt définis.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(f, t)

Get a list of records for a frame and all higher (calling) and lower frames, and the size of the higher part.

format_stack_entry(frame_lineno, lprefix=':')

Return a string with information about a stack entry, identified by a `(frame, lineno)` tuple :

- The canonical form of the filename which contains the frame.
- Le nom de la fonction, ou "<lambda>".
- Les arguments donnés.
- Le résultat.
- La ligne de code (si elle existe).

The following two methods can be called by clients to use a debugger to debug a *statement*, given as a string.

run(cmd, globals=None, locals=None)

Debug a statement executed via the `exec()` function. `globals` defaults to `__main__.__dict__`, `locals` defaults to `globals`.

runeval(expr, globals=None, locals=None)

Debug an expression executed via the `eval()` function. `globals` and `locals` have the same meaning as in `run()`.

runctx(cmd, globals, locals)

For backwards compatibility. Calls the `run()` method.

runcall(func, *args, **kws)

Debug a single function call, and return its result.

Finally, the module defines the following functions :

`bdb.checkfuncname(b, frame)`

Check whether we should break here, depending on the way the breakpoint *b* was set.

If it was set via line number, it checks if `b.line` is the same as the one in the frame also passed as argument. If the breakpoint was set via function name, we have to check we are in the right frame (the right function) and if we are in its first executable line.

`bdb.effective(file, line, frame)`

Determine if there is an effective (active) breakpoint at this line of code. Return a tuple of the breakpoint and a boolean that indicates if it is ok to delete a temporary breakpoint. Return `(None, None)` if there is no matching breakpoint.

`bdb.set_trace()`

Start debugging with a *Bdb* instance from caller's frame.

27.2 faulthandler — Dump the Python traceback

Nouveau dans la version 3.3.

This module contains functions to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal. Call `faulthandler.enable()` to install fault handlers for the SIGSEGV, SIGFPE, SIGABRT, SIGBUS, and SIGILL signals. You can also enable them at startup by setting the PYTHONFAULTHANDLER environment variable or by using the `-X faulthandler` command line option.

The fault handler is compatible with system fault handlers like Apport or the Windows fault handler. The module uses an alternative stack for signal handlers if the `sigaltstack()` function is available. This allows it to dump the traceback even on a stack overflow.

The fault handler is called on catastrophic cases and therefore can only use signal-safe functions (e.g. it cannot allocate memory on the heap). Because of this limitation traceback dumping is minimal compared to normal Python tracebacks :

- Only ASCII is supported. The `backslashreplace` error handler is used on encoding.
- Each string is limited to 500 characters.
- Only the filename, the function name and the line number are displayed. (no source code)
- It is limited to 100 frames and 100 threads.
- The order is reversed : the most recent call is shown first.

By default, the Python traceback is written to `sys.stderr`. To see tracebacks, applications must be run in the terminal. A log file can alternatively be passed to `faulthandler.enable()`.

The module is implemented in C, so tracebacks can be dumped on a crash or when Python is deadlocked.

27.2.1 Dumping the traceback

`faulthandler.dump_traceback(file=sys.stderr, all_threads=True)`

Dump the tracebacks of all threads into *file*. If *all_threads* is `False`, dump only the current thread.

Modifié dans la version 3.5 : Added support for passing file descriptor to this function.

27.2.2 Fault handler state

`faulthandler.enable(file=sys.stderr, all_threads=True)`

Enable the fault handler : install handlers for the SIGSEGV, SIGFPE, SIGABRT, SIGBUS and SIGILL signals to dump the Python traceback. If `all_threads` is `True`, produce tracebacks for every running thread. Otherwise, dump only the current thread.

The `file` must be kept open until the fault handler is disabled : see *issue with file descriptors*.

Modifié dans la version 3.5 : Added support for passing file descriptor to this function.

Modifié dans la version 3.6 : On Windows, a handler for Windows exception is also installed.

`faulthandler.disable()`

Disable the fault handler : uninstall the signal handlers installed by `enable()`.

`faulthandler.is_enabled()`

Check if the fault handler is enabled.

27.2.3 Dumping the tracebacks after a timeout

`faulthandler.dump_traceback_later(timeout, repeat=False, file=sys.stderr, exit=False)`

Dump the tracebacks of all threads, after a timeout of `timeout` seconds, or every `timeout` seconds if `repeat` is `True`. If `exit` is `True`, call `_exit()` with `status=1` after dumping the tracebacks. (Note `_exit()` exits the process immediately, which means it doesn't do any cleanup like flushing file buffers.) If the function is called twice, the new call replaces previous parameters and resets the timeout. The timer has a sub-second resolution.

The `file` must be kept open until the traceback is dumped or `cancel_dump_traceback_later()` is called : see *issue with file descriptors*.

This function is implemented using a watchdog thread and therefore is not available if Python is compiled with threads disabled.

Modifié dans la version 3.5 : Added support for passing file descriptor to this function.

`faulthandler.cancel_dump_traceback_later()`

Cancel the last call to `dump_traceback_later()`.

27.2.4 Dumping the traceback on a user signal

`faulthandler.register(signum, file=sys.stderr, all_threads=True, chain=False)`

Register a user signal : install a handler for the `signum` signal to dump the traceback of all threads, or of the current thread if `all_threads` is `False`, into `file`. Call the previous handler if `chain` is `True`.

The `file` must be kept open until the signal is unregistered by `unregister()` : see *issue with file descriptors*.

Not available on Windows.

Modifié dans la version 3.5 : Added support for passing file descriptor to this function.

`faulthandler.unregister(signum)`

Unregister a user signal : uninstall the handler of the `signum` signal installed by `register()`. Return `True` if the signal was registered, `False` otherwise.

Not available on Windows.

27.2.5 Issue with file descriptors

`enable()`, `dump_traceback_later()` and `register()` keep the file descriptor of their *file* argument. If the file is closed and its file descriptor is reused by a new file, or if `os.dup2()` is used to replace the file descriptor, the traceback will be written into a different file. Call these functions again each time that the file is replaced.

27.2.6 Example

Example of a segmentation fault on Linux with and without enabling the fault handler :

```
$ python3 -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python3 -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

27.3 pdb — Le débogueur Python

Code source : [Lib/pdb.py](#)

Le module `pdb` définit un débogueur de code source interactif pour les programmes Python. Il supporte le paramétrage (conditionnel) de points d'arrêt et l'exécution du code source ligne par ligne, l'inspection des *frames* de la pile, la liste du code source, et l'évaluation arbitraire de code Python dans le contexte de n'importe quelle *frame* de la pile. Il supporte aussi le débogage post-mortem et peut être contrôlé depuis un programme.

Le débogueur est extensible – Il est en réalité défini comme la classe `Pdb`. C'est actuellement non-documenté mais facilement compréhensible en lisant le code source. L'interface d'extension utilise les modules `bdb` et `cmd`.

L'invite du débogueur est `(Pdb)`. L'usage typique pour exécuter un programme sous le contrôle du débogueur est :

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

Modifié dans la version 3.3 : La complétion via le module `readline` est disponible pour les commandes et les arguments de commande, par exemple les noms `*global*` et `*local*` sont proposés comme arguments de la commande ``p``.

Le fichier `pdb.py` peut aussi être invoqué comme un script pour déboguer d'autres scripts. Par exemple :

```
python3 -m pdb myscript.py
```

Si le programme débogué se termine anormalement, *pdb* entrera en débogage post-mortem. Après le débogage post-mortem (ou après une sortie normale du programme), *pdb* redémarrera le programme. Le redémarrage automatique préserve l'état de *pdb* (tels que les points d'arrêt) et dans la plupart des cas est plus utile que de quitter le débogueur à la sortie du programme.

Nouveau dans la version 3.2 : Le fichier `pdb.py` accepte maintenant une option `-c` qui exécute les commandes comme si elles provenaient d'un fichier `.pdbrc`, voir [Commande du débogueur](#).

L'usage typique pour forcer le débogueur depuis un programme s'exécutant est d'insérer

```
import pdb; pdb.set_trace()
```

à l'endroit où vous voulez pénétrer dans le débogueur. Vous pouvez alors parcourir le code suivant cette instruction, et continuer à exécuter sans le débogueur en utilisant la commande *continue*.

L'usage typique pour inspecter un programme planté :

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print(spam)
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print(spam)
(Pdb)
```

Le module définit les fonctions suivantes ; chacune entre dans le débogueur d'une manière légèrement différente :

`pdb.run(statement, globals=None, locals=None)`

Exécute la *déclaration* (donnée sous forme de chaîne de caractères ou d'objet code) sous le contrôle du débogueur. L'invite de débogage apparaît avant l'exécution de tout code ; vous pouvez définir des points d'arrêt et taper *continue*, ou vous pouvez passer à travers l'instruction en utilisant *step* ou *next* (toutes ces commandes sont expliquées ci-dessous). Les arguments *globals* et *locals* optionnels spécifient l'environnement dans lequel le code est exécuté ; par défaut le dictionnaire du module `__main__` est utilisé. (Voir l'explication des fonctions natives `exec()` ou `eval()`.)

`pdb.runeval(expression, globals=None, locals=None)`

Évalue l'*expression* (donnée comme une chaîne de caractères ou un code objet) sous le contrôle du débogueur. Quand la fonction `runeval()` retourne, elle renvoie la valeur de l'expression. Autrement cette fonction est similaire à la fonction `run()`.

`pdb.runcall(function, *args, **kws)`

Appelle la *function* (une fonction ou une méthode, pas une chaîne de caractères) avec les arguments donnés. Quand `runcall()` revient, il retourne ce que l'appel de fonction a renvoyé. L'invite de débogage apparaît dès que la fonction est entrée.

`pdb.set_trace()`

Entre le débogueur dans la *frame* de la pile d'appel. Ceci est utile pour coder en dur un point d'arrêt dans un programme, même si le code n'est pas autrement débogué (par exemple, quand une assertion échoue).

`pdb.post_mortem (traceback=None)`

Entre le débogage post-mortem de l'objet *traceback* donné. Si aucun *traceback* n'est donné, il utilise celui de l'exception en cours de traitement (une exception doit être gérée si la valeur par défaut doit être utilisée).

`pdb.pm ()`

Entre le débogage post-mortem de la trace trouvé dans `sys.last_traceback`.

Les fonctions `run*` et `set_trace()` sont des alias pour instancier la classe *Pdb* et appeler la méthode du même nom. Si vous souhaitez accéder à d'autres fonctionnalités, vous devez le faire vous-même :

class `pdb.Pdb (completekey='tab', stdin=None, stdout=None, skip=None, nosigint=False, readrc=True)`

Le classe du débogueur est la classe *Pdb*.

Les arguments *completekey*, *stdin* et *stdout* sont transmis à la classe sous-jacente `cmd.Cmd`; voir la description ici.

L'argument *skip*, s'il est donné, doit être un itérable des noms de modules de style *glob*. Le débogueur n'entrera pas dans les *frames* qui proviennent d'un module qui correspond à l'un de ces motifs.¹

Par défaut, *Pdb* définit un gestionnaire pour le signal SIGINT (qui est envoyé lorsque l'utilisateur appuie sur `Ctrl-C` sur la console) lorsque vous donnez une commande `continue`. Ceci vous permet de pénétrer à nouveau dans le débogueur en appuyant sur `Ctrl-C`. Si vous voulez que *Pdb* ne touche pas le gestionnaire SIGINT, assignez *nosigint* à *True*.

L'argument *readrc* vaut *True* par défaut et contrôle si *Pdb* chargera les fichiers *.pdbrc* depuis le système de fichiers.

Exemple d'appel pour activer le traçage avec *skip* :

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

Nouveau dans la version 3.1 : L'argument *skip*.

Nouveau dans la version 3.2 : L'argument *nosigint*. Auparavant, un gestionnaire SIGINT n'était jamais configuré par *Pdb*.

Modifié dans la version 3.6 : L'argument *readrc*.

run (*statement*, *globals=None*, *locals=None*)

runeval (*expression*, *globals=None*, *locals=None*)

runcall (*function*, **args*, ***kwargs*)

set_trace ()

Voir la documentation pour les fonctions expliquées ci-dessus.

27.3.1 Commande du débogueur

Les commandes reconnues par le débogueur sont listées. La plupart des commandes peuvent être abrégées à une ou deux lettres comme indiquées; par exemple, `h` (`elp`) signifie que soit `h` ou `help` peut être utilisée pour entrer la commande *help* (mais pas `he` or `hel`, ni `H` ou `HELP`). Les arguments des commandes doivent être séparés par des espaces (espaces ou tabulations). Les arguments optionnels sont entourés dans des crochets (`[]`) dans la syntaxe de la commande; les crochets ne doivent pas être insérés. Les alternatives dans la syntaxe de la commande sont séparés par une barre verticale (`|`).

Entrer une ligne vide répète la dernière commande entrée. Exception : si la dernière commande était la commande *list*, les 11 prochaines lignes sont affichées.

Les commandes que le débogueur ne reconnaît pas sont supposées être des instructions Python et sont exécutées dans le contexte du programme en cours de débogage. Les instructions Python peuvent également être préfixées avec un point d'exclamation (`!`). C'est une façon puissante d'inspecter le programme en cours de débogage; il est même possible de changer une variable ou d'appeler une fonction. Lorsqu'une exception se produit dans une telle instruction, le nom de l'exception est affiché mais l'état du débogueur n'est pas modifié.

Le débogueur supporte *aliases*. Les alias peuvent avoir des paramètres qui permettent un certain niveau d'adaptabilité au contexte étudié.

1. La question de savoir si une *frame* est considérée comme provenant d'un certain module est déterminée par le `__name__` dans les globales de la *frame*.

Plusieurs commandes peuvent être saisies sur une seule ligne, séparées par `;;`. (Un seul `;` n'est pas utilisé car il est le séparateur de plusieurs commandes dans une ligne qui est passée à l'analyseur Python. Aucune intelligence n'est appliquée pour séparer les commandes ; l'entrée est divisée à la première paire de `;;` paire, même si il est au milieu d'une chaîne de caractères.

Si un fichier `.pdbrc` existe dans le répertoire d'accueil de l'utilisateur ou dans le répertoire courant, il est lu et exécuté comme si il avait été écrit dans l'invite du débogueur. C'est particulièrement utile pour les alias. Si les deux fichiers existent, celui dans le répertoire d'accueil de l'utilisateur est lu en premier et les alias définis dedans peuvent être surchargés par le fichier local.

Modifié dans la version 3.2 : Le fichier `.pdbrc` peut maintenant contenir des commandes qui continue le débogage, comme `continue` ou `next`. Précédemment, ces commandes n'avaient aucun effet.

h(elp) [command]

Sans argument, affiche la liste des commandes disponibles. Avec une *commande* comme argument, affiche l'aide de cette commande. `help pdb` affiche la documentation complète (la *docstring* du module `pdb`). Puisque l'argument *command* doit être un identificateur, `help exec` doit être entré pour obtenir de l'aide sur la commande !.

w(here)

Affiche une trace de pile, avec la *frame* le plus récent en bas. Une flèche indique le *frame* courant, qui détermine le contexte de la plupart des commandes.

d(own) [count]

Déplace le niveau de la *frame* courante *count* (par défaut un) vers le bas dans la trace de pile (vers une *frame* plus récente).

u(p) [count]

Déplace le niveau de la *frame* courante *count* (par défaut un) vers le haut dans la trace de pile (vers une *frame* plus ancienne).

b(reak) [(*filename:lineno* | *function*) [, *condition*]]

Avec un argument *lineno*, définit une pause dans le fichier courant. Avec un argument *function*, définit une pause à la première instruction exécutable dans cette fonction. Le numéro de ligne peut être préfixé d'un nom de fichier et d'un deux-points, pour spécifier un point d'arrêt dans un autre fichier (probablement celui qui n'a pas encore été chargé). Le fichier est recherché sur `sys.path`. Notez que chaque point d'arrêt reçoit un numéro auquel se réfèrent toutes les autres commandes de point d'arrêt.

Si un second argument est présent, c'est une expression qui doit évaluer à *True* avant que le point d'arrêt ne soit honoré.

Sans argument, liste tous les arrêts, incluant pour chaque point d'arrêt, le nombre de fois qu'un point d'arrêt a été atteint, le nombre de ignore, et la condition associée le cas échéant.

tbreak [(*filename:lineno* | *function*) [, *condition*]]

Point d'arrêt temporaire, qui est enlevé automatiquement au premier passage. Les arguments sont les mêmes que pour `break`.

cl(ear) [*filename:lineno* | *bpnumber* [*bpnumber* ...]]

Avec un argument *filename:lineno*, efface tous les points d'arrêt sur cette ligne. Avec une liste de numéros de points d'arrêt séparés par un espace, efface ces points d'arrêt. Sans argument, efface tous les points d'arrêt (mais demande d'abord confirmation).

disable [*bpnumber* [*bpnumber* ...]]

Désactive les points d'arrêt indiqués sous la forme d'une liste de numéros de points d'arrêt séparés par un espace. Désactiver un point d'arrêt signifie qu'il ne peut pas interrompre l'exécution du programme, mais à la différence d'effacer un point d'arrêt, il reste dans la liste des points d'arrêt et peut être (ré)activé.

enable [*bpnumber* [*bpnumber* ...]]

Active les points d'arrêt spécifiés.

ignore *bpnumber* [*count*]

Définit le nombre de fois où le point d'arrêt donné sera passé. Si le compte est omis, le compte est mis à 0. Un point

d'arrêt devient actif lorsque le compte est nul. Lorsqu'il n'est pas nul, le comptage est diminué à chaque fois que le point d'arrêt est atteint et que le point d'arrêt n'est pas désactivé et que toute condition associée est évaluée comme vraie.

condition *bpnumber* [*condition*]

Définit une nouvelle *condition* pour le point d'arrêt, une expression qui doit évaluer à *True* avant que le point d'arrêt ne soit honoré. Si *condition* est absente, toute condition existante est supprimée, c'est-à-dire que le point d'arrêt est rendu inconditionnel.

commands [*bpnumber*]

Spécifie une liste de commandes pour le numéro du point d'arrêt *bpnumber*. Les commandes elles-mêmes apparaissent sur les lignes suivantes. Tapez une ligne contenant juste *end* pour terminer les commandes. Un exemple :

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

Pour supprimer toutes les commandes depuis un point d'arrêt, écrivez *commands* suivi immédiatement avec *end* ; ceci supprime les commandes.

Sans argument *bpnumber*, les commandes se réfèrent au dernier point d'arrêt défini.

Vous pouvez utiliser les commandes de point d'arrêt pour redémarrer votre programme. Utilisez simplement la commande continue, ou *step*, ou toute autre commande qui reprend l'exécution.

Spécifie toute commande reprenant l'exécution (actuellement continue, *step*, *next*, *return*, *jump*, *quit* et leurs abréviations) termine la liste des commandes (comme si cette commande était immédiatement suivie de la fin). C'est parce que chaque fois que vous reprenez l'exécution (même avec un simple *next* ou *step*), vous pouvez rencontrer un autre point d'arrêt – qui pourrait avoir sa propre liste de commandes, conduisant à des ambiguïtés sur la liste à exécuter.

Si vous utilisez la commande “silence” dans la liste des commandes, le message habituel concernant l'arrêt à un point d'arrêt n'est pas affiché. Ceci peut être souhaitable pour les points d'arrêt qui doivent afficher un message spécifique et ensuite continuer. Si aucune des autres commandes n'affiche quoi que ce soit, vous ne voyez aucun signe indiquant que le point de rupture a été atteint.

s (tep)

Exécute la ligne en cours, s'arrête à la première occasion possible (soit dans une fonction qui est appelée, soit sur la ligne suivante de la fonction courante).

n (ext)

Continue l'exécution jusqu'à ce que la ligne suivante de la fonction en cours soit atteinte ou qu'elle revienne. (La différence entre *next* et *step* est que *step* s'arrête dans une fonction appelée, tandis que *next* exécute les fonctions appelées à (presque) pleine vitesse, ne s'arrêtant qu'à la ligne suivante dans la fonction courante.)

unt (il) [*lineno*]

Sans argument, continue l'exécution jusqu'à ce que la ligne avec un nombre supérieur au nombre actuel soit atteinte.

Avec un numéro de ligne, continue l'exécution jusqu'à ce qu'une ligne avec un numéro supérieur ou égal à celui-ci soit atteinte. Dans les deux cas, arrête également lorsque la *frame* courante revient.

Modifié dans la version 3.2 : Permet de donner un numéro de ligne explicite.

r (eturn)

Continue l'exécution jusqu'au retour de la fonction courante.

c (ont inue)

Continue l'exécution, seulement s'arrête quand un point d'arrêt est rencontré.

j (ump) *lineno*

Définit la prochaine ligne qui sera exécutée. Uniquement disponible dans la *frame* inférieur. Cela vous permet de revenir en arrière et d'exécuter à nouveau le code, ou de passer en avant pour sauter le code que vous ne voulez pas exécuter.

Il est à noter que tous les sauts ne sont pas autorisés – par exemple, il n'est pas possible de sauter au milieu d'une boucle `for` ou en dehors d'une clause `finally`.

l(**ist**) [*first*[, *last*]]

Liste le code source du fichier courant. Sans arguments, liste 11 lignes autour de la ligne courante ou continue le listing précédant. Avec l'argument `.`, liste 11 lignes autour de la ligne courante. Avec un argument, liste les 11 lignes autour de cette ligne. Avec deux arguments, liste la plage donnée ; si le second argument est inférieur au premier, il est interprété comme un compte.

La ligne en cours dans l'image courante est indiquée par `->`. Si une exception est en cours de débogage, la ligne où l'exception a été initialement levée ou propagée est indiquée par `>>`, si elle diffère de la ligne courante.

Nouveau dans la version 3.2 : Le marqueur `>>`.

ll | *longlist*

Liste le code source de la fonction ou du bloc courant. Les lignes intéressantes sont marquées comme pour *list*.

Nouveau dans la version 3.2.

a(**rgs**)

Affiche la liste d'arguments de la fonction courante.

p *expression*

Évalue l'*expression* dans le contexte courant et affiche sa valeur.

Note : `print()` peut aussi être utilisée, mais n'est pas une commande du débogueur — il exécute la fonction Python *print()*.

pp *expression*

Comme la commande *p*, sauf que la valeur de l'expression est joliment affiché en utilisant le module *pprint*.

whatis *expression*

Affiche le type de l'*expression*.

source *expression*

Essaie d'obtenir le code source pour l'objet donné et l'affiche.

Nouveau dans la version 3.2.

display [*expression*]

Affiche la valeur de l'expression si elle a changée, chaque fois que l'exécution s'arrête dans la *frame* courante.

Sans expression, liste toutes les expressions pour la *frame* courante.

Nouveau dans la version 3.2.

undisplay [*expression*]

N'affiche plus l'expression dans la *frame* courante. Sans expression, efface toutes les expressions d'affichage de la *frame* courante.

Nouveau dans la version 3.2.

interact

Démarre un interpréteur interactif (en utilisant le module *code*) dont l'espace de nommage global contient tous les noms (*global* et *local*) trouvés dans la portée courante.

Nouveau dans la version 3.2.

alias [*name* [*command*]]

Créez un alias appelé *name* qui exécute *command*. La commande ne doit pas être entourée de guillemets. Les paramètres remplaçables peuvent être indiqués par `%1`, `%2` et ainsi de suite, tandis que `%%` est remplacé par tous les paramètres. Si aucune commande n'est donnée, l'alias courant pour *name* est affiché. Si aucun argument n'est donné, tous les alias sont listés.

Les alias peuvent être imbriqués et peuvent contenir tout ce qui peut être légalement tapé à l'invite *pdb*. Notez que les commandes *pdb* internes peuvent être remplacées par des alias. Une telle commande est alors masquée jusqu'à

ce que l'alias soit supprimé. L'*aliasing* est appliqué récursivement au premier mot de la ligne de commande ; tous les autres mots de la ligne sont laissés seuls.

Comme un exemple, voici deux alias utiles (spécialement quand il est placé dans le fichier `.pdbrc`) :

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print("%1.",k,"=",%1.__dict__[k])
# Print instance variables in self
alias ps pi self
```

unalias name

Supprime l'alias spécifié.

! statement

Exécute l'instruction *statement* (une ligne) dans le contexte de la *frame* de la pile courante. Le point d'exclamation peut être omis à moins que le premier mot de l'instruction ne ressemble à une commande de débogueur. Pour définir une variable globale, vous pouvez préfixer la commande d'assignation avec une instruction `global` sur la même ligne, par exemple :

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]

restart [args ...]

Redémarre le programme Python débogué. Si un argument est fourni, il est splitté avec `shlex` et le résultat est utilisé comme le nouveau `sys.argv`. L'historique, les points d'arrêt, les actions et les options du débogueur sont préservés. `restart` est un alias pour `run`.

q(uit)

Quitte le débogueur. Le programme exécuté est arrêté.

Notes

27.4 The Python Profilers

Source code : `Lib/profile.py` and `Lib/pstats.py`

27.4.1 Introduction to the profilers

`cProfile` and `profile` provide *deterministic profiling* of Python programs. A *profile* is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the `pstats` module.

The Python standard library provides two different implementations of the same profiling interface :

1. `cProfile` is recommended for most users ; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on `lsprof`, contributed by Brett Rosen and Ted Czotter.
2. `profile`, a pure Python module whose interface is imitated by `cProfile`, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

Note : The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is `timeit` for reasonably accurate results). This particularly applies to benchmarking Python

code against C code : the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.

27.4.2 Instant User's Manual

This section is provided for users that « don't want to read the manual. » It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do :

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(Use `profile` instead of `cProfile` if the latter is not available on your system.)

The above action would run `re.compile()` and print profile results like the following :

```
197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.001	0.001	<string>:1(<module>)
1	0.000	0.000	0.001	0.001	re.py:212(compile)
1	0.000	0.000	0.001	0.001	re.py:268(_compile)
1	0.000	0.000	0.000	0.000	sre_compile.py:172(_compile_charset)
1	0.000	0.000	0.000	0.000	sre_compile.py:201(_optimize_charset)
4	0.000	0.000	0.000	0.000	sre_compile.py:25(_identityfunction)
3/1	0.000	0.000	0.000	0.000	sre_compile.py:33(_compile)

The first line indicates that 197 calls were monitored. Of those calls, 192 were *primitive*, meaning that the call was not induced via recursion. The next line : `Ordered by: standard name`, indicates that the text string in the far right column was used to sort the output. The column headings include :

ncalls for the number of calls.

tottime for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall is the quotient of `tottime` divided by `ncalls`

cumtime is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate *even* for recursive functions.

percall is the quotient of `cumtime` divided by primitive calls

filename :lineno(function) provides the respective data of each function

When there are two numbers in the first column (for example 3/1), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the `run()` function :

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

The `pstats.Stats` class reads profile results from a file and formats them in various ways.

The file `cProfile` can also be invoked as a script to profile another script. For example :

```
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

`-o` writes the profile results to a file instead of to stdout

`-s` specifies one of the `sort_stats()` sort values to sort the output by. This only applies when `-o` is not supplied.

The `pstats` module's `Stats` class has a variety of methods for manipulating and printing the data saved into a profile results file :

```
import pstats
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The `strip_dirs()` method removed the extraneous path from all the module names. The `sort_stats()` method sorted all the entries according to the standard module/line/name string that is printed. The `print_stats()` method printed out all the statistics. You might try the following sort calls :

```
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with :

```
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do :

```
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try :

```
p.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try :

```
p.sort_stats('time', 'cumulative').print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re : `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do :

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do :

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the `pstats` module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using `cmd`) and interactive help.

27.4.3 `profile` and `cProfile` Module Reference

Both the `profile` and `cProfile` modules provide the following functions :

`profile.run (command, filename=None, sort=-1)`

This function takes a single argument that can be passed to the `exec()` function, and an optional file name. In all cases this routine executes :

```
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a `Stats` instance and prints a simple profiling report. If the sort value is specified, it is passed to this `Stats` instance to control how the results are sorted.

`profile.runctx (command, globals, locals, filename=None, sort=-1)`

This function is similar to `run()`, with added arguments to supply the globals and locals dictionaries for the `command` string. This routine executes :

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the `run()` function above.

class `profile.Profile (timer=None, timeunit=0.0, subcalls=True, builtins=True)`

This class is normally only used if more precise control over profiling is needed than what the `cProfile.run()` function provides.

A custom timer can be supplied for measuring how long code takes to run via the `timer` argument. This must be a function that returns a single number representing the current time. If the number is an integer, the `timeunit` specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the `Profile` class allows formatting profile results without writing the profile data to a file :

```
import cProfile, pstats, io
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

enable()

Start collecting profiling data.

disable()

Stop collecting profiling data.

create_stats()

Stop collecting profiling data and record the results internally as the current profile.

print_stats (sort=-1)

Create a `Stats` object based on the current profile and print the results to stdout.

dump_stats (filename)

Write the results of the current profile to `filename`.

run (cmd)

Profile the `cmd` via `exec()`.

runctx (*cmd, globals, locals*)

Profile the *cmd* via `exec()` with the specified global and local environment.

runcall (*func, *args, **kwargs*)

Profile `func(*args, **kwargs)`

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a `sys.exit()` call during the called command/function execution) no profiling results will be printed.

27.4.4 The Stats Class

Analysis of the profiler data is done using the `Stats` class.

class `pstats.Stats` (**filenames or profile, stream=sys.stdout*)

This class constructor creates an instance of a « statistics object » from a *filename* (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by *stream*.

The file selected by the above constructor must have been created by the corresponding version of `profile` or `cProfile`. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing `Stats` object, the `add()` method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or `profile.Profile` object can be used as the profile data source.

`Stats` objects have the following methods :

strip_dirs ()

This method for the `Stats` class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a « random » order, as it was just after object initialization and loading. If `strip_dirs()` causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add (**filenames*)

This method of the `Stats` class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of `profile.run()` or `cProfile.run()`. Statistics for identically named (re : file, line, name) functions are automatically accumulated into single function statistics.

dump_stats (*filename*)

Save the data loaded into the `Stats` object to a file named *filename*. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the `profile.Profile` and `cProfile.Profile` classes.

sort_stats (**keys*)

This method modifies the `Stats` object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example : 'time' or 'name').

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats('name', 'file')` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined :

Valid Arg	Signification
'calls'	call count
'cumulative'	cumulative time
'cumtime'	cumulative time
'file'	file name
'filename'	file name
'module'	file name
'ncalls'	call count
'pcalls'	primitive call count
'line'	line number
'name'	function name
'nfl'	name/file/line
'stdname'	standard name
'time'	internal time
'tottime'	internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between 'nfl' and 'stdname' is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, 'nfl' does a numeric compare of the line numbers. In fact, `sort_stats('nfl')` is the same as `sort_stats('name', 'file', 'line')`.

For backward-compatibility reasons, the numeric arguments -1, 0, 1, and 2 are permitted. They are interpreted as 'stdname', 'calls', 'time', and 'cumulative' respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

reverse_order()

This method for the *Stats* class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(*restrictions)

This method for the *Stats* class prints out a report as described in the *profile.run()* definition.

The order of the printing is based on the last *sort_stats()* operation done on the object (subject to caveats in *add()* and *strip_dirs()*).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a string that will interpreted as a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example :

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `. *foo:`. In contrast, the command :

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `. *foo:`, and then proceed to only print the first 10% of them.

print_callers(*restrictions)

This method for the *Stats* class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by *print_stats()*, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats :

- With `profile`, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With `cProfile`, each caller is preceded by three numbers : the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

`print_callees (*restrictions)`

This method for the `Stats` class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re : called vs was called by), the arguments and ordering are identical to the `print_callers()` method.

27.4.5 What Is Deterministic Profiling ?

Deterministic profiling is meant to reflect the fact that all *function call*, *function return*, and *exception* events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, *statistical profiling* (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify « hot loops » that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

27.4.6 Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying « clock » is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the « error » will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it « takes a while » from when an event is dispatched until the profiler's call to get the time actually *gets* the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it *can* accumulate and become very significant.

The problem is more important with `profile` than with the lower-overhead `cProfile`. For this reason, `profile` provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-).) Do *not* be alarmed by negative numbers in the profile. They should *only* appear if you have calibrated your profiler, and the results are actually better than without calibration.

27.4.7 Calibration

The profiler of the `profile` module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see [Limitations](#)).

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running Mac OS X, and using Python's `time.clock()` as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it :

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will « less often » show up as negative in profile statistics.

27.4.8 Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the `Profile` class constructor :

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using `profile.Profile` or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently :

`profile.Profile` `your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what `os.times()` returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see [Calibration](#)). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (`os.times()` is *pretty* bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

`cProfile.Profile` `your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example,

if `your_integer_time_func` returns times measured in thousands of seconds, you would construct the `Profile` instance as follows :

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.

Python 3.3 adds several new functions in `time` that can be used to make precise measurements of process or wall-clock time. For example, see `time.perf_counter()`.

27.5 `timeit` — Mesurer le temps d'exécution de fragments de code

Code source : <Lib/timeit.py>

Ce module fournit une façon simple de mesurer le temps d'exécution de fragments de code Python. Il expose une *Interface en ligne de commande* ainsi qu'une *interface Python*. Ce module permet d'éviter un certain nombre de problèmes classiques liés à la mesure des temps d'exécution. Voir par exemple à ce sujet l'introduction par Tim Peters du chapitre « Algorithmes » dans le livre *Python Cookbook*, aux éditions O'Reilly.

27.5.1 Exemples simples

L'exemple suivant illustre l'utilisation de l'*Interface en ligne de commande* afin de comparer trois expressions différentes :

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 23.2 usec per loop
```

L'*Interface Python* peut être utilisée aux mêmes fins avec :

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

Note however that `timeit` will automatically determine the number of repetitions only when the command-line interface is used. In the *Exemples* section you can find more advanced examples.

27.5.2 Interface Python

Ce module définit une classe publique ainsi que trois fonctions destinées à simplifier son usage :

`timeit.timeit(stmt='pass', setup='pass', timer=<default timer>, number=1000000, globals=None)`

Crée une instance d'objet `Timer` à partir de l'instruction donnée, du code `setup` et de la fonction `timer`, puis exécute sa méthode `timeit()` à `number` reprises. L'argument optionnel `globals` spécifie un espace de nommage dans lequel exécuter le code.

Modifié dans la version 3.5 : Le paramètre optionnel `globals` a été ajouté.

`timeit.repeat(stmt='pass', setup='pass', timer=<default timer>, repeat=3, number=1000000, globals=None)`

Crée une instance d'objet `Timer` à partir de l'instruction donnée, du code `setup` et de la fonction `timer`, puis exécute sa méthode `repeat()` à `number` reprises, `repeat` fois. L'argument optionnel `globals` spécifie un espace de nommage dans lequel exécuter le code.

Modifié dans la version 3.5 : Le paramètre optionnel `globals` a été ajouté.

`timeit.default_timer()`

Le minuteur par défaut, qui est toujours `time.perf_counter()`.

Modifié dans la version 3.3 : `time.perf_counter()` est désormais le minuteur par défaut.

class `timeit.Timer(stmt='pass', setup='pass', timer=<timer function>, globals=None)`

Classe permettant de mesurer le temps d'exécution de fragments de code.

Ce constructeur prend en argument une instruction dont le temps d'exécution doit être mesuré, une instruction additionnelle de mise en place et une fonction de chronométrage. Les deux instructions valent `'pass'` par défaut ; la fonction de chronométrage dépend de la plateforme d'exécution (se référer au *doc string* du module). `stmt` et `setup` peuvent contenir plusieurs instructions séparées par des `;` ou des sauts de lignes tant qu'ils ne comportent pas de littéraux sur plusieurs lignes. L'instruction est exécutée dans l'espace de nommage de `timeit` par défaut ; ce comportement peut être modifié en passant un espace de nommage au paramètre `globals`.

Pour mesurer le temps d'exécution de la première instruction, utilisez la méthode `timeit()`. Les méthodes `repeat()` et `autorange()` sont des méthodes d'agrément permettant d'appeler `timeit()` à plusieurs reprises.

Le temps d'exécution de `setup` n'est pas pris en compte dans le temps global d'exécution.

Les paramètres `stmt` et `setup` peuvent également recevoir des objets appelables sans argument. Ceci transforme alors les appels à ces objets en fonction de chronométrage qui seront exécutées par `timeit()`. Notez que le surcoût lié à la mesure du temps d'exécution dans ce cas est légèrement supérieur en raisons des appels de fonction supplémentaires.

Modifié dans la version 3.5 : Le paramètre optionnel `globals` a été ajouté.

`timeit(number=1000000)`

Mesure le temps `number` exécution de l'instruction principale. Ceci exécute l'instruction de mise en place une seule fois puis renvoie un flottant correspondant au temps nécessaire à l'exécution de l'instruction principale à plusieurs reprises, mesuré en secondes. L'argument correspond au nombre d'itérations dans la boucle, par défaut un million. L'instruction principale, l'instruction de mise en place et la fonction de chronométrage utilisées sont passées au constructeur.

Note : By default, `timeit()` temporarily turns off *garbage collection* during the timing. The advantage of this approach is that it makes independent timings more comparable. This disadvantage is that GC may be an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the `setup` string. For example :

```
timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
```

autorange (`callback=None`)

Détermine automatiquement combien de fois appeler `timeit()`.

This is a convenience function that calls `timeit()` repeatedly so that the total time ≥ 0.2 second, returning the eventual (number of loops, time taken for that number of loops). It calls `timeit()` with *number* set to successive powers of ten (10, 100, 1000, ...) up to a maximum of one billion, until the time taken is at least 0.2 second, or the maximum is reached.

Si *callback* est spécifié et n'est pas `None`, elle est appelée après chaque itération avec deux arguments (numéro de l'itération et temps écoulé) : `callback(number, time_taken)`.

Nouveau dans la version 3.6.

repeat (*repeat*=3, *number*=1000000)

Appelle `timeit()` plusieurs fois.

Cette fonction d'agrément appelle `timeit()` à plusieurs reprises et renvoie une liste de résultats. Le premier argument spécifie le nombre d'appels à `timeit()`. Le second argument spécifie l'argument *number* de `timeit()`.

Note : Il est tentant de vouloir calculer la moyenne et l'écart-type des résultats et notifier ces valeurs. Ce n'est cependant pas très utile. En pratique, la valeur la plus basse donne une estimation basse de la vitesse maximale à laquelle votre machine peut exécuter le fragment de code spécifié ; les valeurs hautes de la liste sont typiquement provoquées non pas par une variabilité de la vitesse d'exécution de Python, mais par d'autres processus interférant avec la précision du chronométrage. Le `min()` du résultat est probablement la seule valeur à laquelle vous devriez vous intéresser. Pour aller plus loin, vous devriez regarder l'intégralité des résultats et utiliser le bon sens plutôt que les statistiques.

print_exc (*file*=None)

Outil permettant d'afficher la trace du code chronométré.

Usage typique :

```
t = Timer(...)          # outside the try/except
try:
    t.timeit(...)        # or t.repeat(...)
except Exception:
    t.print_exc()
```

L'avantage par rapport à la trace standard est que les lignes sources du code compilé sont affichées. Le paramètre optionnel *file* définit l'endroit où la trace est envoyée, par défaut `sys.stderr`.

27.5.3 Interface en ligne de commande

Lorsque le module est appelé comme un programme en ligne de commande, la syntaxe suivante est utilisée :

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-t] [-c] [-h] [statement ...]
```

Les options suivantes sont gérées :

- n** N, **--number**=N
nombre d'exécutions de l'instruction *statement*
 - r** N, **--repeat**=N
how many times to repeat the timer (default 3)
 - s** S, **--setup**=S
instruction exécutée une seule fois à l'initialisation (pass par défaut)
 - p**, **--process**
mesure le temps au niveau du processus et non au niveau du système, en utilisant `time.process_time()` plutôt que `time.perf_counter()` qui est utilisée par défaut
- Nouveau dans la version 3.3.

-t, --time
use `time.time()` (deprecated)

-u, --unit=U
specify a time unit for timer output; can select usec, msec, or sec
Nouveau dans la version 3.5.

-c, --clock
use `time.clock()` (deprecated)

-v, --verbose
affiche les temps d'exécutions bruts, répéter pour plus de précision

-h, --help
affiche un court message d'aide puis quitte

Une instruction sur plusieurs lignes peut être donnée en entrée en spécifiant chaque ligne comme un argument séparé. Indenter une ligne est possible en encadrant l'argument de guillemets et en le préfixant par des espaces. Plusieurs `-s` sont gérées de la même façon.

Si `-n` n'est pas donnée, le nombre de boucles adapté est déterminé automatiquement en essayant les puissances de 10 successives jusqu'à ce que le temps total d'exécution dépasse 0,2 secondes.

`default_timer()` measurements can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The `-r` option is good for this; the default of 3 repetitions is probably enough in most cases. You can use `time.process_time()` to measure CPU time.

Note : Il existe un surcoût minimal associé à l'exécution de l'instruction `pass`. Le code présenté ici ne tente pas de le masquer, mais vous devez être conscient de son existence. Ce surcoût minimal peut être mesuré en invoquant le programme sans argument; il peut différer en fonction des versions de Python.

27.5.4 Exemples

Il est possible de fournir une instruction de mise en place exécutée une seule fois au début du chronométrage :

```
$ python -m timeit -s 'text = "sample string"; char = "g"' 'char in text'
10000000 loops, best of 3: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"' 'text.find(char)'
1000000 loops, best of 3: 0.342 usec per loop
```

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

La même chose peut être réalisée en utilisant la classe `Timer` et ses méthodes :

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40193588800002544, 0.3960157959998014, 0.39594301399984033]
```


Les exemples qui suivent montrent comment chronométrer des expressions sur plusieurs lignes. Nous comparons ici le coût d'utilisation de `hasattr()` par rapport à `try/except` pour tester la présence ou l'absence d'attributs d'un objet :

```
$ python -m timeit 'try: ' ' str.__bool__ 'except AttributeError: ' ' pass'
100000 loops, best of 3: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__bool__"): pass'
100000 loops, best of 3: 4.26 usec per loop

$ python -m timeit 'try: ' ' int.__bool__ 'except AttributeError: ' ' pass'
1000000 loops, best of 3: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__bool__"): pass'
100000 loops, best of 3: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

Afin de permettre à `timeit` d'accéder aux fonctions que vous avez définies, vous pouvez passer au paramètre `setup` une instruction d'importation :

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Une autre possibilité est de passer `globals()` au paramètre `globals`, ceci qui exécutera le code dans l'espace de nommage global courant. Cela peut être plus pratique que de spécifier manuellement des importations :

```
def f(x):
    return x**2
def g(x):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

27.6 `trace` — Trace or track Python statement execution

Code source : [Lib/abc.py](#)

The `trace` module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.

Voir aussi :

Coverage.py A popular third-party coverage tool that provides HTML output along with advanced features such as branch coverage.

27.6.1 Utilisation en ligne de commande.

The `trace` module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute `somefile.py` and generate annotated listings of all Python modules imported during the execution into the current directory.

--help

Display usage and exit.

--version

Display the version of the module and exit.

Main options

At least one of the following options must be specified when invoking `trace`. The `--listfuncs` option is mutually exclusive with the `--trace` and `--count` options. When `--listfuncs` is provided, neither `--count` nor `--trace` are accepted, and vice versa.

-c, --count

Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also `--coverdir`, `--file` and `--no-report` below.

-t, --trace

Display lines as they are executed.

-l, --listfuncs

Display the functions executed by running the program.

- r, --report**
Produce an annotated list from an earlier program run that used the `--count` and `--file` option. This does not execute any code.
- T, --trackcalls**
Display the calling relationships exposed by running the program.

Modifiers

- f, --file=<file>**
Name of a file to accumulate counts over several tracing runs. Should be used with the `--count` option.
- C, --coverdir=<dir>**
Directory where the report files go. The coverage report for `package.module` is written to file `dir/package/module.cover`.
- m, --missing**
When generating annotated listings, mark lines which were not executed with `>>>>>>`.
- s, --summary**
When using `--count` or `--report`, write a brief summary to stdout for each file processed.
- R, --no-report**
Do not generate annotated listings. This is useful if you intend to make several runs with `--count`, and then produce a single set of annotated listings at the end.
- g, --timing**
Prefix each line with the time since the program started. Only used while tracing.

Filters

These options may be repeated multiple times.

- ignore-module=<mod>**
Ignore each of the given module names and its submodules (if it is a package). The argument can be a list of names separated by a comma.
- ignore-dir=<dir>**
Ignore all modules and packages in the named directory and subdirectories. The argument can be a list of directories separated by `os.pathsep`.

27.6.2 Programmatic Interface

class `trace.Trace` (*count=1, trace=1, countfuncs=0, countcallers=0, ignoremods=(), ignoredirs=(), infile=None, outfile=None, timing=False*)

Create an object to trace execution of a single statement or expression. All parameters are optional. *count* enables counting of line numbers. *trace* enables line execution tracing. *countfuncs* enables listing of the functions called during the run. *countcallers* enables call relationship tracking. *ignoremods* is a list of modules or packages to ignore. *ignoredirs* is a list of directories whose modules or packages should be ignored. *infile* is the name of the file from which to read stored count information. *outfile* is the name of the file in which to write updated count information. *timing* enables a timestamp relative to when tracing was started to be displayed.

run (*cmd*)

Execute the command and gather statistics from the execution with the current tracing parameters. *cmd* must be a string or code object, suitable for passing into `exec()`.

runctx (*cmd*, *globals=None*, *locals=None*)

Execute the command and gather statistics from the execution with the current tracing parameters, in the defined global and local environments. If not defined, *globals* and *locals* default to empty dictionaries.

runfunc (*func*, **args*, ***kwargs*)

Call *func* with the given arguments under control of the *Trace* object with the current tracing parameters.

results ()

Return a *CoverageResults* object that contains the cumulative results of all previous calls to *run*, *runctx* and *runfunc* for the given *Trace* instance. Does not reset the accumulated trace results.

class *trace.CoverageResults*

A container for coverage results, created by *Trace.results()*. Should not be created directly by the user.

update (*other*)

Merge in data from another *CoverageResults* object.

write_results (*show_missing=True*, *summary=False*, *coverdir=None*)

Write coverage results. Set *show_missing* to show lines that had no hits. Set *summary* to include in the output the coverage summary per module. *coverdir* specifies the directory into which the coverage result files will be output. If *None*, the results for each source file are placed in its directory.

A simple example demonstrating the use of the programmatic interface :

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main() ')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

27.7 tracemalloc — Trace memory allocations

Nouveau dans la version 3.4.

Source code : [Lib/tracemalloc.py](#)

The *tracemalloc* module is a debug tool to trace memory blocks allocated by Python. It provides the following information :

- Traceback where an object was allocated
- Statistics on allocated memory blocks per filename and per line number : total size, number and average size of allocated memory blocks
- Compute the differences between two snapshots to detect memory leaks

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the `PYTHONTRACEMALLOC` environment variable to 1, or by using `-X tracemalloc` command line option. The `tracemalloc.start()` function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup : set the `PYTHONTRACEMALLOC` environment variable to 25, or use the `-X tracemalloc=25` command line option.

27.7.1 Exemples

Display the top 10

Display the 10 files allocating the most memory :

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output of the Python test suite :

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108.
↪B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

We can see that Python loaded 4855 KiB data (bytecode and constants) from modules and that the `collections` module allocated 244 KiB to build `namedtuple` types.

See `Snapshot.statistics()` for more options.

Compute differences

Take two snapshots and display the differences :

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite :

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369), ↵
↪average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106), ↵
↪average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589), ↵
↪average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526), ↵
↪average=139 B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334), ↵
↪average=86 B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1), ↵
↪average=96.0 KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109), ↵
↪average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143), ↵
↪average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969), ↵
↪average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126), ↵
↪average=546 B
```

We can see that Python has loaded 8173 KiB of module data (bytecode and constants), and that this is 4428 KiB more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the `linecache` module has cached 940 KiB of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the `Snapshot.dump()` method to analyze the snapshot offline. Then use the `Snapshot.load()` method reload the snapshot.

Get the traceback of a memory block

Code to display the traceback of the biggest memory block :

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Example of output of the Python test suite (traceback limited to 25 frames) :

```
903 memory blocks: 870.1 KiB
File "<frozen importlib._bootstrap>", line 716
File "<frozen importlib._bootstrap>", line 1036
File "<frozen importlib._bootstrap>", line 934
File "<frozen importlib._bootstrap>", line 1068
File "<frozen importlib._bootstrap>", line 619
File "<frozen importlib._bootstrap>", line 1581
File "<frozen importlib._bootstrap>", line 1614
File "/usr/lib/python3.4/doctest.py", line 101
import pdb
File "<frozen importlib._bootstrap>", line 284
File "<frozen importlib._bootstrap>", line 938
File "<frozen importlib._bootstrap>", line 1068
File "<frozen importlib._bootstrap>", line 619
File "<frozen importlib._bootstrap>", line 1581
File "<frozen importlib._bootstrap>", line 1614
File "/usr/lib/python3.4/test/support/__init__.py", line 1728
import doctest
File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
File "/usr/lib/python3.4/test/regtest.py", line 1276
    test_runner()
File "/usr/lib/python3.4/test/regtest.py", line 976
    display_failure=not verbose)
File "/usr/lib/python3.4/test/regtest.py", line 761
    match_tests=ns.match_tests)
File "/usr/lib/python3.4/test/regtest.py", line 1563
    main()
File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

We can see that the most memory was allocated in the `importlib` module to load data (bytecode and constants) from

modules : 870.1 KiB. The traceback is where the `importlib` loaded data most recently : on the `import pdb` line of the `doctest` module. The traceback may change if a new module is loaded.

Pretty top

Code to display the 10 lines allocating the most memory with a pretty output, ignoring `<frozen importlib._bootstrap>` and `<unknown>` files :

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        # replace "/path/to/module/file.py" with "module/file.py"
        filename = os.sep.join(frame.filename.split(os.sep)[-2:])
        print("#%s: %s: %s: %.1f KiB"
              % (index, filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Example of output of the Python test suite :

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
   _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
   _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
   exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
   cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
   testMethod()
```

(suite sur la page suivante)

(suite de la page précédente)

```
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

See `Snapshot.statistics()` for more options.

27.7.2 API

Fonctions

`tracemalloc.clear_traces()`

Clear traces of memory blocks allocated by Python.

See also `stop()`.

`tracemalloc.get_object_traceback(obj)`

Get the traceback where the Python object *obj* was allocated. Return a `Traceback` instance, or `None` if the `tracemalloc` module is not tracing memory allocations or did not trace the allocation of the object.

See also `gc.get_referrers()` and `sys.getsizeof()` functions.

`tracemalloc.get_traceback_limit()`

Get the maximum number of frames stored in the traceback of a trace.

The `tracemalloc` module must be tracing memory allocations to get the limit, otherwise an exception is raised.

The limit is set by the `start()` function.

`tracemalloc.get_traced_memory()`

Get the current size and peak size of memory blocks traced by the `tracemalloc` module as a tuple: (current: int, peak: int).

`tracemalloc.get_tracemalloc_memory()`

Get the memory usage in bytes of the `tracemalloc` module used to store traces of memory blocks. Return an `int`.

`tracemalloc.is_tracing()`

True if the `tracemalloc` module is tracing Python memory allocations, False otherwise.

See also `start()` and `stop()` functions.

`tracemalloc.start(nframe: int=1)`

Start tracing Python memory allocations: install hooks on Python memory allocators. Collected tracebacks of traces will be limited to *nframe* frames. By default, a trace of a memory block only stores the most recent frame: the limit is 1. *nframe* must be greater or equal to 1.

Storing more than 1 frame is only useful to compute statistics grouped by 'traceback' or to compute cumulative statistics: see the `Snapshot.compare_to()` and `Snapshot.statistics()` methods.

Storing more frames increases the memory and CPU overhead of the `tracemalloc` module. Use the `get_tracemalloc_memory()` function to measure how much memory is used by the `tracemalloc` module.

The `PYTHONTRACEMALLOC` environment variable (`PYTHONTRACEMALLOC=NFRAME`) and the `-X tracemalloc=NFRAME` command line option can be used to start tracing at startup.

See also `stop()`, `is_tracing()` and `get_traceback_limit()` functions.

`tracemalloc.stop()`

Stop tracing Python memory allocations : uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call `take_snapshot()` function to take a snapshot of traces before clearing them.

See also `start()`, `is_tracing()` and `clear_traces()` functions.

`tracemalloc.take_snapshot()`

Take a snapshot of traces of memory blocks allocated by Python. Return a new `Snapshot` instance.

The snapshot does not include memory blocks allocated before the `tracemalloc` module started to trace memory allocations.

Tracebacks of traces are limited to `get_traceback_limit()` frames. Use the `nframe` parameter of the `start()` function to store more frames.

The `tracemalloc` module must be tracing memory allocations to take a snapshot, see the `start()` function.

See also the `get_object_traceback()` function.

DomainFilter

class `tracemalloc.DomainFilter` (*inclusive* : *bool*, *domain* : *int*)

Filter traces of memory blocks by their address space (domain).

Nouveau dans la version 3.6.

inclusive

If *inclusive* is `True` (include), match memory blocks allocated in the address space *domain*.

If *inclusive* is `False` (exclude), match memory blocks not allocated in the address space *domain*.

domain

Address space of a memory block (*int*). Read-only property.

Filter

class `tracemalloc.Filter` (*inclusive* : *bool*, *filename_pattern* : *str*, *lineno* : *int*=*None*, *all_frames* : *bool*=*False*, *domain* : *int*=*None*)

Filter on traces of memory blocks.

See the `fnmatch.fnmatch()` function for the syntax of *filename_pattern*. The `'.pyc'` file extension is replaced with `'.py'`.

Exemples :

— `Filter(True, subprocess.__file__)` only includes traces of the `subprocess` module

— `Filter(False, tracemalloc.__file__)` excludes traces of the `tracemalloc` module

— `Filter(False, "<unknown>")` excludes empty tracebacks

Modifié dans la version 3.5 : The `'.pyo'` file extension is no longer replaced with `'.py'`.

Modifié dans la version 3.6 : Added the `domain` attribute.

domain

Address space of a memory block (*int* or *None*).

inclusive

If *inclusive* is `True` (include), only match memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

lineno

Line number (*int*) of the filter. If *lineno* is *None*, the filter matches any line number.

filename_pattern

Filename pattern of the filter (*str*). Read-only property.

all_frames

If *all_frames* is `True`, all frames of the traceback are checked. If *all_frames* is `False`, only the most recent frame is checked.

This attribute has no effect if the traceback limit is 1. See the `get_traceback_limit()` function and `Snapshot.traceback_limit` attribute.

Frame**class** tracemalloc.**Frame**

Frame of a traceback.

The `Traceback` class is a sequence of `Frame` instances.

filename

Filename (str).

lineno

Line number (int).

Snapshot**class** tracemalloc.**Snapshot**

Snapshot of traces of memory blocks allocated by Python.

The `take_snapshot()` function creates a snapshot instance.

compare_to (old_snapshot : Snapshot, key_type : str, cumulative : bool=False)

Compute the differences with an old snapshot. Get statistics as a sorted list of `StatisticDiff` instances grouped by *key_type*.

See the `Snapshot.statistics()` method for *key_type* and *cumulative* parameters.

The result is sorted from the biggest to the smallest by : absolute value of `StatisticDiff.size_diff`, `StatisticDiff.size`, absolute value of `StatisticDiff.count_diff`, `StatisticDiff.count` and then by `StatisticDiff.traceback`.

dump (filename)

Write the snapshot into a file.

Use `load()` to reload the snapshot.

filter_traces (filters)

Create a new `Snapshot` instance with a filtered `traces` sequence, *filters* is a list of `DomainFilter` and `Filter` instances. If *filters* is an empty list, return a new `Snapshot` instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

Modifié dans la version 3.6 : `DomainFilter` instances are now also accepted in *filters*.

classmethod load (filename)

Load a snapshot from a file.

See also `dump()`.

statistics (key_type : str, cumulative : bool=False)

Get statistics as a sorted list of `Statistic` instances grouped by *key_type* :

key_type	description
'filename'	filename
'lineno'	filename and line number
'traceback'	traceback

If *cumulative* is `True`, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with *key_type* equals to 'filename' and 'lineno'.

The result is sorted from the biggest to the smallest by : *Statistic.size*, *Statistic.count* and then by *Statistic.traceback*.

traceback_limit

Maximum number of frames stored in the traceback of *traces* : result of the *get_traceback_limit()* when the snapshot was taken.

traces

Traces of all memory blocks allocated by Python : sequence of *Trace* instances.

The sequence has an undefined order. Use the *Snapshot.statistics()* method to get a sorted list of statistics.

Statistic

class tracemalloc.**Statistic**

Statistic on memory allocations.

Snapshot.statistics() returns a list of *Statistic* instances.

See also the *StatisticDiff* class.

count

Number of memory blocks (int).

size

Total size of memory blocks in bytes (int).

traceback

Traceback where the memory block was allocated, *Traceback* instance.

StatisticDiff

class tracemalloc.**StatisticDiff**

Statistic difference on memory allocations between an old and a new *Snapshot* instance.

Snapshot.compare_to() returns a list of *StatisticDiff* instances. See also the *Statistic* class.

count

Number of memory blocks in the new snapshot (int) : 0 if the memory blocks have been released in the new snapshot.

count_diff

Difference of number of memory blocks between the old and the new snapshots (int) : 0 if the memory blocks have been allocated in the new snapshot.

size

Total size of memory blocks in bytes in the new snapshot (int) : 0 if the memory blocks have been released in the new snapshot.

size_diff

Difference of total size of memory blocks in bytes between the old and the new snapshots (int) : 0 if the memory blocks have been allocated in the new snapshot.

traceback

Traceback where the memory blocks were allocated, *Traceback* instance.

Trace

class tracemalloc.**Trace**

Trace of a memory block.

The *Snapshot.traces* attribute is a sequence of *Trace* instances.

size

Size of the memory block in bytes (int).

traceback

Traceback where the memory block was allocated, *Traceback* instance.

Traceback

class tracemalloc.**Traceback**

Sequence of *Frame* instances sorted from the most recent frame to the oldest frame.

A traceback contains at least 1 frame. If the *tracemalloc* module failed to get a frame, the filename "<unknown>" at line number 0 is used.

When a snapshot is taken, tracebacks of traces are limited to *get_traceback_limit()* frames. See the *take_snapshot()* function.

The *Trace.traceback* attribute is an instance of *Traceback* instance.

format (*limit=None*)

Format the traceback as a list of lines with newlines. Use the *linecache* module to retrieve lines from the source code. If *limit* is set, only format the *limit* most recent frames.

Similar to the *traceback.format_tb()* function, except that *format()* does not include newlines.

Exemple :

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

Sortie :

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

Paquets et distribution de paquets logiciels

Ces bibliothèques vous aident lors de la publication et l'installation de logiciels Python. Bien que ces modules sont conçus pour fonctionner avec le [Python Package Index](#), ils peuvent aussi être utilisés avec un serveur local, ou sans serveur.

28.1 `distutils` — Création et installation des modules Python

Le package `distutils` fournit le support pour la création et l'installation de modules supplémentaires dans une installation Python. Les nouveaux modules peuvent être soit en Python pur à 100%, soit des modules d'extension écrits en C, soit des collections de paquets Python qui incluent des modules codés en C et en Python.

La plupart des utilisateurs de Python ne voudront *pas* utiliser ce module directement, mais plutôt les outils cross-version maintenus par la *Python Packaging Authority*. En particulier, `setuptools` est une alternative améliorée à `distutils` qui fournit :

- support pour la déclaration des dépendances de projets
- mécanismes supplémentaires pour configurer quels fichiers inclure dans les distributions source (y compris les extensions pour l'intégration avec les systèmes de contrôle de version)
- la possibilité de déclarer les « points d'entrée » du projet, qui peuvent être utilisés comme base pour les systèmes d'extensions
- la possibilité de générer automatiquement des exécutables en ligne de commande Windows au moment de l'installation plutôt que de devoir les pré-construire
- comportement cohérent entre toutes les versions Python supportées

Le programme d'installation recommandé `pip` exécute tous les scripts `setup.py` avec `setuptools`, même si le script lui-même n'importe que `distutils`. Pour plus d'informations, reportez-vous au [Python Packaging User Guide](#).

À destination des auteurs et utilisateurs d'outils d'emballage cherchant une compréhension plus approfondie des détails du système actuel de création de paquets et de leur distribution, la documentation utilisateur historique de `distutils` la référence de son API restent disponibles :

- `install-index`
- `distutils-index`

28.2 ensurepip — Bootstrapping the pip installer

Nouveau dans la version 3.4.

The `ensurepip` package provides support for bootstrapping the `pip` installer into an existing Python installation or virtual environment. This bootstrapping approach reflects the fact that `pip` is an independent project with its own release cycle, and the latest available stable version is bundled with maintenance and feature releases of the CPython reference interpreter.

In most cases, end users of Python shouldn't need to invoke this module directly (as `pip` should be bootstrapped by default), but it may be needed if installing `pip` was skipped when installing Python (or when creating a virtual environment) or after explicitly uninstalling `pip`.

Note : This module *does not* access the internet. All of the components needed to bootstrap `pip` are included as internal parts of the package.

Voir aussi :

installing-index The end user guide for installing Python packages

PEP 453 : Explicit bootstrapping of pip in Python installations The original rationale and specification for this module.

28.2.1 Command line interface

The command line interface is invoked using the interpreter's `-m` switch.

The simplest possible invocation is :

```
python -m ensurepip
```

This invocation will install `pip` if it is not already installed, but otherwise does nothing. To ensure the installed version of `pip` is at least as recent as the one bundled with `ensurepip`, pass the `--upgrade` option :

```
python -m ensurepip --upgrade
```

By default, `pip` is installed into the current virtual environment (if one is active) or into the system site packages (if there is no active virtual environment). The installation location can be controlled through two additional command line options :

- `--root <dir>` : Installs `pip` relative to the given root directory rather than the root of the currently active virtual environment (if any) or the default root for the current Python installation.
- `--user` : Installs `pip` into the user site packages directory rather than globally for the current Python installation (this option is not permitted inside an active virtual environment).

By default, the scripts `pipX` and `pipX.Y` will be installed (where `X.Y` stands for the version of Python used to invoke `ensurepip`). The scripts installed can be controlled through two additional command line options :

- `--altinstall` : if an alternate installation is requested, the `pipX` script will *not* be installed.
- **`--default-pip` : if a « default pip » installation is requested, the `pip` script will be installed in addition to the two regular scripts.**

Providing both of the script selection options will trigger an exception.

Modifié dans la version 3.6.3 : The exit status is non-zero if the command fails.

28.2.2 Module API

`ensurepip` exposes two functions for programmatic use :

`ensurepip.version()`

Returns a string specifying the bundled version of `pip` that will be installed when bootstrapping an environment.

`ensurepip.bootstrap (root=None, upgrade=False, user=False, altinstall=False, default_pip=False, verbosity=0)`

Bootstraps `pip` into the current or designated environment.

`root` specifies an alternative root directory to install relative to. If `root` is `None`, then installation uses the default install location for the current environment.

`upgrade` indicates whether or not to upgrade an existing installation of an earlier version of `pip` to the bundled version.

`user` indicates whether to use the user scheme rather than installing globally.

By default, the scripts `pipX` and `pipX.Y` will be installed (where `X.Y` stands for the current version of Python).

If `altinstall` is set, then `pipX` will *not* be installed.

If `default_pip` is set, then `pip` will be installed in addition to the two regular scripts.

Setting both `altinstall` and `default_pip` will trigger `ValueError`.

`verbosity` controls the level of output to `sys.stdout` from the bootstrapping operation.

Note : The bootstrapping process has side effects on both `sys.path` and `os.environ`. Invoking the command line interface in a subprocess instead allows these side effects to be avoided.

Note : The bootstrapping process may install additional modules required by `pip`, but other software should not assume those dependencies will always be present by default (as the dependencies may be removed in a future version of `pip`).

28.3 venv — Création d’environnements virtuels

Nouveau dans la version 3.3.

Code source : [Lib/venv/](#)

The `venv` module provides support for creating lightweight « virtual environments » with their own site directories, optionally isolated from system site directories. Each virtual environment has its own Python binary (allowing creation of environments with various Python versions) and can have its own independent set of installed Python packages in its site directories.

Voir la [PEP 405](#) pour plus d’informations à propos des environnements virtuels Python.

Note : Le script `pyenv` est obsolète depuis Python 3.6 et a été remplacé par `python3 -m venv`.

28.3.1 Création d'environnements virtuels

La création d'*environnements virtuels* est faite en exécutant la commande `venv` :

```
python3 -m venv /path/to/new/virtual/environment
```

Running this command creates the target directory (creating any parent directories that don't exist already) and places a `pyvenv.cfg` file in it with a `home` key pointing to the Python installation from which the command was run. It also creates a `bin` (or `Scripts` on Windows) subdirectory containing a copy/symlink of the Python binary/binaries (as appropriate for the platform or arguments used at environment creation time). It also creates an (initially empty) `lib/pythonX.Y/site-packages` subdirectory (on Windows, this is `Lib\site-packages`). If an existing directory is specified, it will be re-used.

Obsolète depuis la version 3.6 : `pyvenv` était l'outil recommandé pour créer des environnements sous Python 3.3 et 3.4, et est [obsolète depuis Python 3.6](#).

Modifié dans la version 3.5 : L'utilisation de `venv` est maintenant recommandée pour créer vos environnements virtuels.

Voir aussi :

[Guide Utilisateur de l'Empaquetage Python : Créer et utiliser des environnements virtuels](#)

Sur Windows, appelez la commande `venv` comme suit :

```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

Alternativement, si vous avez configuré les variables `PATH` et `PATHEXT` pour votre installation Python :

```
c:\>python -m venv c:\path\to\myenv
```

La commande, si lancée avec `-h`, montrera les options disponibles :

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
           [--upgrade] [--without-pip]
           ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR                A directory to create the environment in.

optional arguments:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
  --symlinks            Try to use symlinks rather than copies, when symlinks
                        are not the default for the platform.
  --copies              Try to use copies rather than symlinks, even when
                        symlinks are the default for the platform.
  --clear              Delete the contents of the environment directory if it
                        already exists, before environment creation.
  --upgrade            Upgrade the environment directory to use this version
                        of Python, assuming Python has been upgraded in-place.
  --without-pip        Skips installing or upgrading pip in the virtual
                        environment (pip is bootstrapped by default)

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

Modifié dans la version 3.4 : Installe pip par défaut, ajout des options `--without-pip` et `--copies`

Modifié dans la version 3.4 : Dans les versions précédentes, si le dossier de destination existait déjà, une erreur était levée, sauf si l'option `--clear` ou `--upgrade` était incluse.

Le fichier crée `pyenv.cfg` inclus aussi la clé `include-system-site-packages`, dont la valeur est `true` si `venv` est lancé avec l'option `--system-site-packages`, sinon sa valeur est `false`.

Sauf si l'option `--without-pip` est incluse, `ensurepip` sera invoqué pour amorcer pip dans l'environnement virtuel.

Plusieurs chemins peuvent être donnés à `venv`, et dans ce cas un environnement virtuel sera créé, en fonction des options incluses, à chaque chemin donné.

Once a virtual environment has been created, it can be « activated » using a script in the virtual environment's binary directory. The invocation of the script is platform-specific (<venv> must be replaced by the path of the directory containing the virtual environment) :

Plateforme	Invite de commande	Commande pour activer l'environnement virtuel
Posix	bash/zsh	\$ source <venv>/bin/activate
	fish	\$. <venv>/bin/activate.fish
	csh/tcsh	\$ source <venv>/bin/activate.csh
Windows	cmd.exe	C:\> <venv>\Scripts\activate.bat
	PowerShell	PS C:> <venv>\Scripts\Activate.ps1

Vous ne devez pas spécialement activer un environnement ; l'activation ajoute juste le chemin du dossier de binaires de votre environnement virtuel à votre PATH, pour que « python » invoque l'interpréteur Python de l'environnement virtuel et que vous puissiez lancer des scripts installés sans avoir à utiliser leur chemin complet. Cependant, tous les scripts installés dans un environnement virtuel devraient être exécutables sans l'activer, et se lancer avec l'environnement virtuel Python automatiquement.

Vous pouvez désactiver un environnement virtuel en écrivant « **deactivate** » dans votre shell. Le mécanisme est spécifique à chaque plateforme : Par exemple, le script d'activation pour Bash définit une fonction « **deactivate** », alors que sur Windows il y a des scripts séparés appelés `deactivate.bat` et `Deactivate.ps1` qui sont installés quand l'environnement virtuel est créé.

Nouveau dans la version 3.4 : Les scripts d'activation pour `fish` et `csh`.

Note : Un environnement virtuel est un environnement Python tel que l'interpréteur Python, les bibliothèques et les scripts installés sont isolés de ceux installés dans d'autres environnements virtuels, et (par défaut) de toutes autres bibliothèques installées dans un Python « système », par exemple celui qui est installé avec votre système d'exploitation.

Un environnement virtuel est une arborescence de dossiers qui contiennent les fichiers exécutables Python et autres fichiers qui indiquent que c'est un environnement virtuel.

Les outils d'installations communs comme `Setuptools` et `pip` fonctionnent comme prévu avec des environnements virtuels. En d'autres termes, quand un environnement virtuel est actif, ils installent les paquets Python dans l'environnement virtuel sans avoir besoin de leur préciser explicitement.

Quand un environnement virtuel est actif (Par exemple quand l'interpréteur Python de l'environnement virtuel est lancé), les attributs `sys.prefix` et `sys.exec_prefix` pointent vers le dossier racine de l'environnement virtuel, alors que `sys.base_prefix` et `sys.base_exec_prefix` pointent vers l'installation de Python qui n'est pas celle de l'environnement virtuel et qui a été utilisée pour créer l'environnement virtuel. Si un environnement virtuel n'est pas actif, alors `sys.prefix` est égal à `sys.base_prefix` et `sys.exec_prefix` est égal à `sys.base_exec_prefix` (ils pointent tous sur une installation Python qui n'est pas un environnement virtuel).

Quand un environnement virtuel est actif, toute option qui change le chemin d'installation sera ignoré de tous les fichiers de configuration **distutils** pour éviter que des projets soient accidentellement installés en dehors de l'environnement virtuel.

Quand vous travaillez dans une invite de commande **shell**, les utilisateurs peuvent activer un environnement virtuel en lançant un script `activate` situé dans le dossier des exécutables de l'environnement virtuel (le nom de fichier précis dépend du shell utilisé), ce qui ajoute le dossier des exécutables de l'environnement virtuel dans la variable d'environnement `PATH`. Il ne devrait jamais y avoir besoin dans d'autre circonstances d'activer un environnement virtuel. Des scripts installés dans un environnement virtuel ont un **shebang** qui pointe vers l'interpréteur Python de l'environnement virtuel. Cela veut dire que le script sera lancé avec cet interpréteur peut importe la valeur de `PATH`. Sur Windows, le **shebang** est interprété si vous avez le Lanceur Python pour Windows installé (Cela a été ajouté à Python 3.3 – Voir [PEP 397](#) pour plus de détails). De ce fait, double cliquer un script installé dans une fenêtre de l'Explorateur Windows devrait lancer le script avec le bon interpréteur sans avoir besoin de référencer son environnement virtuel dans `PATH`.

28.3.2 API

La méthode haut niveau décrite au dessus utilise une API simple qui permet à des créateurs d'environnements virtuels externes de personnaliser la création d'environnements virtuels basés sur leurs besoins, la classe `EnvBuilder`.

class `venv.EnvBuilder` (*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*)

La classe `EnvBuilder` accepte les arguments suivants lors de l'instanciation :

- `system_site_packages` – Une valeur booléenne qui indique que les site-packages du système Python devraient être disponibles dans l'environnement virtuel (par défaut à `False`).
- `clear` – Une valeur booléenne qui, si vraie, supprimera le contenu de n'importe quel dossier existant cible, avant de créer l'environnement.
- `symlinks` – a Boolean value indicating whether to attempt to symlink the Python binary (and any necessary DLLs or other binaries, e.g. `pythonw.exe`), rather than copying.
- `upgrade` – Une valeur booléenne qui, si vraie, mettra à jour un environnement existant avec le Python lancé – utilisé quand Python a été mis à jour sur place (par défaut à `False`).
- `with_pip` – Une valeur booléenne qui, si vraie, assure que pip est installé dans l'environnement virtuel. Cela utilise `ensurepip` avec l'option `--default-pip`.
- `prompt` – Une chaîne utilisée après que l'environnement virtuel est activé (par défaut à `None` ce qui veut dire qu'il utilisera le nom du dossier de l'environnement).

Modifié dans la version 3.4 : Ajout du paramètre `with_pip`

Nouveau dans la version 3.6 : Ajout du paramètre `prompt`

Les créateurs des outils de création d'environnement virtuel externes seront libres d'utiliser la classe `EnvBuilder` en tant que classe de base.

Le **env-builder** retourné est un objet qui a une méthode, `create` :

create (*env_dir*)

Cette méthode prends en argument obligatoire le chemin (absolu ou relatif par rapport au dossier courant) du dossier cible qui doit contenir l'environnement virtuel. La méthode `create` doit soit créer un environnement dans le dossier spécifié, soit lever une exception.

La méthode `create` de la classe `EnvBuilder` illustre les points d'entrées disponibles pour la personnalisation de sous-classes :

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

Chacune des méthodes `ensure_directories()`, `create_configuration()`, `setup_python()`, `setup_scripts()` et `post_setup()` peuvent être écrasés.

ensure_directories (*env_dir*)

Crée un dossier d'environnement et tous les dossiers nécessaires, et retourne un objet contexte. C'est juste un conteneur pour des attributs (comme des chemins), qui sera utilisé par d'autres méthodes. Ces dossiers peuvent déjà exister. tant que `clear` ou `upgrade` ont été spécifiés pour permettre de telles opérations dans un dossier d'environnement existant.

create_configuration (*context*)

Crée le fichier de configuration `pyenv.cfg` dans l'environnement.

setup_python (*context*)

Creates a copy of the Python executable (and, under Windows, DLLs) in the environment. On a POSIX system, if a specific executable `python3.x` was used, symlinks to `python` and `python3` will be created pointing to that executable, unless files with those names already exist.

setup_scripts (*context*)

Installe les scripts d'activation appropriés à la plateforme dans l'environnement virtuel.

post_setup (*context*)

Une méthode qui n'est là que pour se faire surcharger dans des implémentations externes pour pré installer des paquets dans l'environnement virtuel ou pour exécuter des étapes post-crétation.

De plus, `EnvBuilder` propose cette méthode utilitaire qui peut être appelée de `setup_scripts()` ou `post_setup()` dans des sous classes pour assister dans l'installation de scripts custom dans l'environnement virtuel.

install_scripts (*context*, *path*)

path correspond au chemin vers le dossier qui contient les sous dossiers « **common** », « **posix** », « **nt** », chacun contenant des scripts destinés pour le dossier « **bin** » dans l'environnement. Le contenu du dossier « **common** » et le dossier correspondant à `os.name` sont copiés après quelque remplacement de texte temporaires :

- `__VENV_DIR__` est remplacé avec le chemin absolu du dossier de l'environnement.
- `__VENV_NAME__` est remplacé avec le nom de l'environnement (le dernier segment du chemin vers le dossier de l'environnement).
- `__VENV_PROMPT__` est remplacé par le prompt (nom de l'environnement entouré de parenthèses et avec un espace le suivant).
- `__VENV_BIN_NAME__` est remplacé par le nom du dossier **bin** (soit `bin` soit `Scripts`).
- `__VENV_PYTHON__` est remplacé avec le chemin absolu de l'exécutable de l'environnement.

Les dossiers peuvent exister (pour quand un environnement existant est mis à jour).

Il y a aussi une fonction pratique au niveau du module :

`venv.create(env_dir, system_site_packages=False, clear=False, symlinks=False, with_pip=False)`

Crée une `EnvBuilder` avec les arguments donnés, et appelle sa méthode `create()` avec l'argument `env_dir`.

Modifié dans la version 3.4 : Ajout du paramètre `with_pip`

28.3.3 Un exemple d'extension de `EnvBuilder`

Le script qui suis montre comment étendre `EnvBuilder` en implémentant une sous-classe qui installe **setuptools** et **pip** dans un environnement créé :

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv
```

(suite sur la page suivante)

(suite de la page précédente)

```

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If True, setuptools and pip are not installed into the
        created virtual environment.
    :param nopip: If True, pip is not installed into the created
        virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
        installation can be monitored by passing a progress
        callable. If specified, it is called with two
        arguments: a string indicating some progress, and a
        context indicating where the string is coming from.
        The context argument can have one of three values:
        'main', indicating that it is called from virtualize()
        itself, and 'stdout' and 'stderr', which are obtained
        by reading lines from the output streams of a subprocess
        which is used to install the app.

        If a callable is not specified, default progress
        information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        Set up any packages which need to be pre-installed into the
        virtual environment being created.

        :param context: The information for the virtual environment
            creation request being processed.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
            # Can't install pip without setuptools
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either pass to a progress
        callable (if specified) or write progress information to sys.stderr.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break

```

(suite sur la page suivante)

(suite de la page précédente)

```

        if progress is not None:
            progress(s, context)
        else:
            if not self.verbose:
                sys.stderr.write('.')
            else:
                sys.stderr.write(s.decode('utf-8'))
            sys.stderr.flush()
    stream.close()

def install_script(self, context, name, url):
    _, _, path, _, _, _ = urlparse(url)
    fn = os.path.split(path)[-1]
    binpath = context.bin_path
    distpath = os.path.join(binpath, fn)
    # Download script into the virtual environment's binaries folder
    urlretrieve(url, distpath)
    progress = self.progress
    if self.verbose:
        term = '\n'
    else:
        term = ''
    if progress is not None:
        progress('Installing %s ...%s' % (name, term), 'main')
    else:
        sys.stderr.write('Installing %s ...%s' % (name, term))
        sys.stderr.flush()
    # Install in the virtual environment
    args = [context.env_exe, fn]
    p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
    t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
    t1.start()
    t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
    t2.start()
    p.wait()
    t1.join()
    t2.join()
    if progress is not None:
        progress('done.', 'main')
    else:
        sys.stderr.write('done.\n')
    # Clean up - no longer needed
    os.unlink(distpath)

def install_setuptools(self, context):
    """
    Install setuptools in the virtual environment.

    :param context: The information for the virtual environment
                    creation request being processed.
    """
    url = 'https://bitbucket.org/pypa/setuptools/downloads/ez_setup.py'
    self.install_script(context, 'setuptools', url)
    # clear up the setuptools archive which gets downloaded
    pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
    files = filter(pred, os.listdir(context.bin_path))
    for f in files:

```

(suite sur la page suivante)

(suite de la page précédente)

```

        f = os.path.join(context.bin_path, f)
        os.unlink(f)

def install_pip(self, context):
    """
    Install pip in the virtual environment.

    :param context: The information for the virtual environment
                    creation request being processed.
    """
    url = 'https://raw.githubusercontent.com/pypa/pip/master/contrib/get-pip.py'
    self.install_script(context, 'pip', url)

def main(args=None):
    compatible = True
    if sys.version_info < (3, 3):
        compatible = False
    elif not hasattr(sys, 'base_prefix'):
        compatible = False
    if not compatible:
        raise ValueError('This script is only for use with '
                          'Python 3.3 or later')
    else:
        import argparse

        parser = argparse.ArgumentParser(prog=__name__,
                                         description='Creates virtual Python '
                                                         'environments in one or '
                                                         'more target '
                                                         'directories.')
        parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                            help='A directory in which to create the '
                                  'virtual environment.')
        parser.add_argument('--no-setuptools', default=False,
                            action='store_true', dest='nodist',
                            help="Don't install setuptools or pip in the "
                                  "virtual environment.")
        parser.add_argument('--no-pip', default=False,
                            action='store_true', dest='nopip',
                            help="Don't install pip in the virtual "
                                  "environment.")
        parser.add_argument('--system-site-packages', default=False,
                            action='store_true', dest='system_site',
                            help='Give the virtual environment access to the '
                                  'system site-packages dir.')

        if os.name == 'nt':
            use_symlinks = False
        else:
            use_symlinks = True
        parser.add_argument('--symlinks', default=use_symlinks,
                            action='store_true', dest='symlinks',
                            help='Try to use symlinks rather than copies, '
                                  'when symlinks are not the default for '
                                  'the platform.')
        parser.add_argument('--clear', default=False, action='store_true',
                            dest='clear', help='Delete the contents of the '
                                                  'virtual environment ')

```

(suite sur la page suivante)

(suite de la page précédente)

```

        'directory if it already '
        'exists, before virtual '
        'environment creation.')
    parser.add_argument('--upgrade', default=False, action='store_true',
                        dest='upgrade', help='Upgrade the virtual '
                        'environment directory to '
                        'use this version of '
                        'Python, assuming Python '
                        'has been upgraded '
                        'in-place.')
    parser.add_argument('--verbose', default=False, action='store_true',
                        dest='verbose', help='Display the output '
                        'from the scripts which '
                        'install setuptools and pip.')

    options = parser.parse_args(args)
    if options.upgrade and options.clear:
        raise ValueError('you cannot supply --upgrade and --clear together.')
    builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                clear=options.clear,
                                symlinks=options.symlinks,
                                upgrade=options.upgrade,
                                nodist=options.nodist,
                                nopip=options.nopip,
                                verbose=options.verbose)

    for d in options.dirs:
        builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)

```

This script is also available for download [online](#).

28.4 zipapp — Manage executable python zip archives

Nouveau dans la version 3.5.

Code source : [Lib/zipapp.py](#)

Ce module fournit des outils pour gérer la création de fichiers zip contenant du code Python, qui peuvent être exécutés directement par l'interpréteur Python. Le module fournit à la fois une interface de ligne de commande *Interface en ligne de commande* et une interface *API Python*.

28.4.1 Exemple de base

L'exemple suivant montre comment l'interface de ligne de commande *Interface en ligne de commande* peut être utilisée pour créer une archive exécutable depuis un répertoire contenant du code Python. Lors de l'exécution, l'archive exécutera la fonction `main` du module `myapp` dans l'archive.

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

28.4.2 Interface en ligne de commande

Lorsqu'il est appelé en tant que programme à partir de la ligne de commande, la syntaxe suivante est utilisée :

```
$ python -m zipapp source [options]
```

Si *source* est un répertoire, une archive est créée à partir du contenu de *source*. Si *source* est un fichier, ce doit être une archive et il est copié dans l'archive cible (ou le contenu de sa ligne *shebang* est affiché si l'option `--info` est indiquée).

Les options suivantes sont disponibles :

-o <output>, **--output**=<output>

Écrit la sortie dans un fichier nommé *output*. Si cette option n'est pas spécifiée, le nom du fichier de sortie sera le même que celui de l'entrée *source*, avec l'extension `.pyz`. Si un nom de fichier explicite est donné, il est utilisé tel quel (une extension `.pyz` doit donc être incluse si nécessaire).

Un nom de fichier de sortie doit être spécifié si la *source* est une archive (et, dans ce cas, la *sortie* ne doit pas être la même que la *source*).

-p <interpreter>, **--python**=<interpreter>

Ajoute une ligne `#!` à l'archive en spécifiant *interpreter* comme commande à exécuter. Aussi, sur un système POSIX, cela rend l'archive exécutable. Le comportement par défaut est de ne pas écrire la ligne `#!` et de ne pas rendre le fichier exécutable.

-m <mainfn>, **--main**=<mainfn>

Écrit un fichier `__main__.py` dans l'archive qui exécute *mainfn*. L'argument *mainfn* est de la forme « *pkg.mod:fn* », où « *pkg.mod* » est un paquet/module dans l'archive, et « *fn* » est un callable dans le module donné. Le fichier `__main__.py` réalise cet appel.

`--main` ne peut pas être spécifié lors de la copie d'une archive.

--info

Affiche l'interpréteur intégré dans l'archive, à des fins de diagnostic. Dans ce cas, toutes les autres options sont ignorées et *SOURCE* doit être une archive et non un répertoire.

-h, **--help**

Affiche un court message d'aide et quitte.

28.4.3 API Python

Ce module définit deux fonctions utilitaires :

`zipapp.create_archive(source, target=None, interpreter=None, main=None)`

Crée une archive d'application à partir de *source*. La source peut être de natures suivantes :

- The name of a directory, or a `pathlib.Path` object referring to a directory, in which case a new application archive will be created from the content of that directory.
- The name of an existing application archive file, or a `pathlib.Path` object referring to such a file, in which case the file is copied to the target (modifying it to reflect the value given for the *interpreter* argument). The file name should include the `.pyz` extension, if required.
- Un objet fichier ouvert pour la lecture en mode binaire. Le contenu du fichier doit être une archive d'application et Python suppose que l'objet fichier est positionné au début de l'archive.

L'argument *target* détermine où l'archive résultante sera écrite :

- If it is the name of a file, or a `pathlib.Path` object, the archive will be written to that file.
- S'il s'agit d'un objet fichier ouvert, l'archive sera écrite dans cet objet fichier, qui doit être ouvert pour l'écriture en mode octets.
- Si la cible est omise (ou `None`), la source doit être un répertoire et la cible sera un fichier portant le même nom que la source, avec une extension `.pyz` ajoutée.

L'argument *interpreter* spécifie le nom de l'interpréteur Python avec lequel l'archive sera exécutée. Il est écrit dans une ligne *shebang* au début de l'archive. Sur un système POSIX, cela est interprété par le système d'exploitation et, sur Windows, il sera géré par le lanceur Python. L'omission de l'*interpreter* n'entraîne pas l'écriture d'une ligne *shebang*. Si un interpréteur est spécifié et que la cible est un nom de fichier, le bit exécutable du fichier cible sera mis à 1.

L'argument *main* spécifie le nom d'un callable, utilisé comme programme principal pour l'archive. Il ne peut être spécifié que si la source est un répertoire et si la source ne contient pas déjà un fichier `__main__.py`. L'argument *main* doit prendre la forme `pkg.module:callable` et l'archive sera exécutée en important `pkg.module` et en exécutant l'appelable donné sans argument. Omettre *main* est une erreur si la source est un répertoire et ne contient pas un fichier `__main__.py` car, dans ce cas, l'archive résultante ne serait pas exécutable.

Si un objet fichier est spécifié pour *source* ou *target*, il est de la responsabilité de l'appelant de le fermer après avoir appelé `create_archive`.

Lors de la copie d'une archive existante, les objets fichier fournis n'ont besoin que des méthodes `read` et `readline` ou `write`. Lors de la création d'une archive à partir d'un répertoire, si la cible est un objet fichier, elle sera passée à la classe `zipfile.ZipFile` et devra fournir les méthodes nécessaires à cette classe.

`zipapp.get_interpreter(archive)`

Renvoie l'interpréteur spécifié dans la ligne `#!` au début de l'archive. S'il n'y a pas de ligne `#!`, renvoie `None`. L'argument *archive* peut être un nom de fichier ou un objet de type fichier ouvert à la lecture en mode binaire. Python suppose qu'il est au début de l'archive.

28.4.4 Exemples

Regroupe le contenu d'un répertoire dans une archive, puis l'exécute.

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

The same can be done using the `create_archive()` function :

```
>>> import zipapp
>>> zipapp.create_archive('myapp.pyz', 'myapp')
```

Pour rendre l'application directement exécutable sur un système POSIX, spécifiez un interpréteur à utiliser.

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

Pour remplacer la ligne *shebang* sur une archive existante, créez une archive modifiée en utilisant la fonction `create_archive()` :

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

Pour mettre à jour le fichier sans créer de copie locale, effectuez le remplacement en mémoire à l'aide d'un objet `BytesIO`, puis écrasez la source par la suite. Notez qu'il y a un risque lors de l'écrasement d'un fichier local qu'une erreur entraîne la perte du fichier original. Ce code ne protège pas contre de telles erreurs, assurez-vous de prendre les mesures nécessaires en production. De plus, cette méthode ne fonctionnera que si l'archive tient en mémoire :

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

28.4.5 Spécification de l'interprète

Notez que si vous spécifiez un interpréteur et que vous distribuez ensuite votre archive d'application, vous devez vous assurer que l'interpréteur utilisé est portable. Le lanceur Python pour Windows gère la plupart des formes courantes de la ligne POSIX `#!`, mais il y a d'autres problèmes à considérer :

- Si vous utilisez `/usr/bin/env python` (ou d'autres formes de la commande *python*, comme `/usr/bin/python`), vous devez considérer que vos utilisateurs peuvent avoir Python 2 ou Python 3 par défaut, et écrire votre code pour fonctionner dans les deux versions.
- Si vous utilisez une version explicite, par exemple `/usr/bin/env python3` votre application ne fonctionnera pas pour les utilisateurs qui n'ont pas cette version. (C'est peut-être ce que vous voulez si vous n'avez pas rendu votre code compatible Python 2).
- Il n'y a aucun moyen de dire « python X.Y ou supérieur » donc faites attention si vous utilisez une version exacte comme `/usr/bin/env python3.4` car vous devrez changer votre ligne *shebang* pour les utilisateurs de Python 3.5, par exemple.

Normalement, vous devriez utiliser un `/usr/bin/env python2` ou `/usr/bin/env python3`, selon que votre code soit écrit pour Python 2 ou 3.

28.4.6 Création d'applications autonomes avec *zipapp*

En utilisant le module *zipapp*, il est possible de créer des programmes Python qui peuvent être distribués à des utilisateurs finaux dont le seul pré-requis est d'avoir la bonne version de Python installée sur leur ordinateur. Pour y arriver, la clé est de regrouper toutes les dépendances de l'application dans l'archive avec le code source de l'application.

Les étapes pour créer une archive autonome sont les suivantes :

1. Créez votre application dans un répertoire comme d'habitude, de manière à avoir un répertoire `myapp` contenant un fichier `__main__.py` et tout le code de l'application correspondante.
2. Installez toutes les dépendances de votre application dans le répertoire `myapp` en utilisant *pip* :

```
$ python -m pip install -r requirements.txt --target myapp
```

(ceci suppose que vous ayez vos dépendances de projet dans un fichier `requirements.txt` — sinon vous pouvez simplement lister les dépendances manuellement sur la ligne de commande `pip`).

3. Si nécessaire, supprimez les répertoires `.dist-info` créés par `pip` dans le répertoire `myapp`. Ceux-ci contiennent des métadonnées pour `pip` afin de gérer les paquets et, comme vous n'utiliserez plus `pip`, ils ne sont pas nécessaires (c'est sans conséquence si vous les laissez).
4. Regroupez le tout à l'aide de :

```
$ python -m zipapp -p "interpreter" myapp
```

Cela produira un exécutable autonome qui peut être exécuté sur n'importe quelle machine avec l'interpréteur approprié disponible. Voir *Spécification de l'interprète* pour plus de détails. Il peut être envoyé aux utilisateurs sous la forme d'un seul fichier.

Sous Unix, le fichier `myapp.pyz` est exécutable tel quel. Vous pouvez renommer le fichier pour supprimer l'extension `.pyz` si vous préférez un nom de commande « simple ». Sous Windows, le fichier `myapp.pyz[w]` est exécutable en vertu du fait que l'interpréteur Python est associé aux extensions de fichier `.pyz` et `.pyzw` une fois installé.

Création d'un exécutable Windows

Sous Windows, l'association de Python à l'extension `.pyz` est facultative et, de plus, il y a certains mécanismes qui ne reconnaissent pas les extensions enregistrées de manière « transparente » (l'exemple le plus simple est que `subprocess.run(['myapp'])` ne trouvera pas votre application — vous devez explicitement spécifier l'extension).

Sous Windows, il est donc souvent préférable de créer un exécutable à partir du `zipapp`. C'est relativement facile bien que cela nécessite un compilateur C. L'astuce repose sur le fait que les fichiers `zip` peuvent avoir des données arbitraires au début et les fichiers `exe` de Windows peuvent avoir des données arbitraires à la fin. Ainsi, en créant un lanceur approprié et en rajoutant le fichier `.pyz` à sa fin, vous obtenez un fichier unique qui exécute votre application.

Un lanceur approprié peut être aussi simple que ce qui suit :

```
#define Py_LIMITED_API 1
#include "Python.h"

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#ifdef WINDOWS
int WINAPI wWinMain(
    HINSTANCE hInstance,      /* handle to current instance */
    HINSTANCE hPrevInstance,  /* handle to previous instance */
    LPWSTR lpCmdLine,         /* pointer to command line */
    int nCmdShow              /* show state of window */
)
#else
int wmain()
#endif
{
    wchar_t **myargv = _alloca((__argc + 1) * sizeof(wchar_t*));
    myargv[0] = __wargv[0];
    memcpy(myargv + 1, __wargv, __argc * sizeof(wchar_t *));
    return Py_Main(__argc+1, myargv);
}
```

Si vous définissez le symbole du préprocesseur `WINDOWS` cela va générer un exécutable IUG, et sans lui, un exécutable console.

Pour compiler l'exécutable, vous pouvez soit simplement utiliser les outils standards en ligne de commande `MSVC`, soit profiter du fait que `distutils` sait comment compiler les sources Python :

```
>>> from distutils.ccompiler import new_compiler
>>> import distutils.sysconfig
>>> import sys
>>> import os
>>> from pathlib import Path

>>> def compile(src):
>>>     src = Path(src)
>>>     cc = new_compiler()
>>>     exe = src.stem
>>>     cc.add_include_dir(distutils.sysconfig.get_python_inc())
>>>     cc.add_library_dir(os.path.join(sys.base_exec_prefix, 'libs'))
>>>     # First the CLI executable
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe)
>>>     # Now the GUI executable
>>>     cc.define_macro('WINDOWS')
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe + '.w')

>>> if __name__ == "__main__":
>>>     compile("zastub.c")
```

Le lanceur résultant utilise le « Limited ABI » donc il fonctionnera sans changement avec n'importe quelle version de Python 3.x. Tout ce dont il a besoin est que Python (`python3.dll`) soit sur le PATH de l'utilisateur.

Pour une distribution entièrement autonome vous pouvez distribuer le lanceur avec votre application en fin de fichier, empaqueté avec la distribution *embedded* Python. Ceci fonctionnera sur n'importe quel ordinateur avec l'architecture appropriée (32 bits ou 64 bits).

Mises en garde

Il y a certaines limites à l'empaquetage de votre application dans un seul fichier. Dans la plupart des cas, si ce n'est tous, elles peuvent être traitées sans qu'il soit nécessaire d'apporter de modifications majeures à votre application.

1. Si votre application dépend d'un paquet qui inclut une extension C, ce paquet ne peut pas être exécuté à partir d'un fichier zip (c'est une limitation du système d'exploitation, car le code exécutable doit être présent dans le système de fichiers pour que le lanceur de l'OS puisse le charger). Dans ce cas, vous pouvez exclure cette dépendance du fichier zip et, soit demander à vos utilisateurs de l'installer, soit la fournir avec votre fichier zip et ajouter du code à votre fichier `__main__.py` pour inclure le répertoire contenant le module décompressé dans `sys.path`. Dans ce cas, vous devrez vous assurer d'envoyer les binaires appropriés pour votre ou vos architecture(s) cible(s) (et éventuellement choisir la bonne version à ajouter à `sys.path` au moment de l'exécution, basée sur la machine de l'utilisateur).
2. Si vous livrez un exécutable Windows comme décrit ci-dessus, vous devez vous assurer que vos utilisateurs ont `python3.dll` sur leur PATH (ce qui n'est pas le comportement par défaut de l'installateur) ou vous devez inclure la distribution intégrée dans votre application.
3. Le lanceur suggéré ci-dessus utilise l'API d'intégration Python. Cela signifie que dans votre application `sys.executable` sera votre application et *pas* un interpréteur Python classique. Votre code et ses dépendances doivent être préparés à cette possibilité. Par exemple, si votre application utilise le module `multiprocessing`, elle devra appeler `multiprocessing.set_executable()` pour que le module sache où trouver l'interpréteur Python standard.

28.4.7 Le format d'archive d'application Zip Python

Python est capable d'exécuter des fichiers zip qui contiennent un fichier `__main__.py` depuis la version 2.6. Pour être exécutée par Python, une archive d'application doit simplement être un fichier zip standard contenant un fichier `__main__.py` qui sera exécuté comme point d'entrée de l'application. Comme d'habitude pour tout script Python, le parent du script (dans ce cas le fichier zip) sera placé sur `sys.path` et ainsi d'autres modules pourront être importés depuis le fichier zip.

Le format de fichier zip permet d'ajouter des données arbitraires à un fichier zip. Le format de l'application zip utilise cette possibilité pour préfixer une ligne *shebang* POSIX standard dans le fichier (`#!/path/to/interpreter`).

Formellement, le format d'application zip de Python est donc :

1. Une ligne *shebang* facultative, contenant les caractères `b'#!` suivis d'un nom d'interpréteur, puis un caractère fin de ligne (`b'\n'`). Le nom de l'interpréteur peut être n'importe quoi acceptable pour le traitement *shebang* de l'OS, ou le lanceur Python sous Windows. L'interpréteur doit être encodé en UTF-8 sous Windows, et en `sys.getfilesystemencoding()` sur POSIX.
2. Des données *zipfile* standards, telles que générées par le module `zipfile`. Le contenu du fichier zip *doit* inclure un fichier appelé `__main__.py` (qui doit se trouver à la racine du fichier zip — c'est-à-dire qu'il ne peut se trouver dans un sous-répertoire). Les données du fichier zip peuvent être compressées ou non.

Si une archive d'application a une ligne *shebang*, elle peut avoir le bit exécutable activé sur les systèmes POSIX, pour lui permettre d'être exécutée directement.

Vous pouvez créer des archives d'applications sans utiliser les outils de ce module — le module existe pour faciliter les choses, mais les archives, créées par n'importe quel moyen tout en respectant le format ci-dessus, sont valides pour Python.

Environnement d'exécution Python

Les modules décrits dans ce chapitre fournissent une large collection de services relatifs à l'interpréteur Python et son interaction avec son environnement. En voici un survol :

29.1 `sys` — Paramètres et fonctions propres à des systèmes

Ce module fournit un accès à certaines variables utilisées et maintenues par l'interpréteur, et à des fonctions interagissant fortement avec ce dernier. Ce module est toujours disponible.

`sys.abiflags`

Contient, sur les systèmes POSIX où Python a été compilé avec le script `configure`, les *ABI flags* tels que définis par la [PEP 3149](#).

Nouveau dans la version 3.2.

`sys.argv`

La liste des arguments de la ligne de commande passés à un script Python. `argv[0]` est le nom du script (chemin complet, ou non, en fonction du système d'exploitation). Si la commande a été exécutée avec l'option `-c` de l'interpréteur, `argv[0]` vaut la chaîne `'-c'`. Si aucun nom de script n'a été donné à l'interpréteur Python, `argv[0]` sera une chaîne vide.

Pour boucler sur l'entrée standard, ou la liste des fichiers donnés sur la ligne de commande, utilisez le module `fileinput`.

`sys.base_exec_prefix`

Défini au démarrage de Python, avant que `site.py` ne soit évalué, à la même valeur que `exec_prefix`. Hors d'un *environnement virtuel*, les valeurs restent les mêmes ; si `site.py` détecte qu'un environnement virtuel est utilisé, les valeurs de `prefix` et `exec_prefix` sont modifiées pour pointer vers l'environnement virtuel, alors que `base_prefix` et `base_exec_prefix` pointent toujours à la racine de l'installation de Python (celui utilisé pour créer l'environnement virtuel).

Nouveau dans la version 3.3.

sys.base_prefix

Défini au démarrage de Python, avant que `site.py` ne soit évalué, à la même valeur que `prefix`. Hors d'un *environnement virtuel*, les valeurs restent les mêmes ; si `site.py` détecte qu'un environnement virtuel est utilisé, les valeurs de `prefix` et `exec_prefix` sont modifiées pour pointer vers l'environnement virtuel, alors que `base_prefix` et `base_exec_prefix` pointent toujours à la racine de l'installation de Python (celui utilisé pour créer l'environnement virtuel).

Nouveau dans la version 3.3.

sys.byteorder

Un indicateur de l'ordre natif des octets. Vaudra `'big'` sur les plateformes gros-boutistes (octet le plus significatif en premier), et `'little'` sur les plateformes petit-boutiste (octet le moins significatif en premier).

sys.builtin_module_names

Un *tuple* de chaînes de caractères donnant les noms de tous les modules compilés dans l'interpréteur Python. (Cette information n'est pas disponible autrement — `modules.keys()` liste seulement les modules importés.)

sys.call_tracing (*func, args*)

Appelle `func(*args)`, avec le traçage activé. L'état du traçage est sauvegardé et restauré après l'appel. Ceci est destiné à être appelé depuis un débogueur à partir d'un point de contrôle, pour déboguer récursivement un autre code.

sys.copyright

Une chaîne contenant le copyright relatif à l'interpréteur Python.

sys._clear_type_cache()

Vide le cache interne de types. Le cache de types est utilisé pour accélérer les recherches d'attributs et de méthodes. N'utilisez cette fonction *que* pour libérer des références inutiles durant le débogage de fuite de référence.

Cette fonction ne devrait être utilisée que pour un usage interne et spécialisé.

sys._current_frames()

Renvoie un dictionnaire faisant correspondre chaque identifiant de fil d'exécution à la *stack frame* actuellement active pour ces fils d'exécution au moment où la fonction est appelée. Notez que les fonctions du module `traceback` peuvent construire une *call stack* à partir d'une telle *frame*.

N'ayant pas besoin de la coopération des fils d'exécution bloqués, cette fonction est très utile pour déboguer un *deadlock*. Aussi, les *call stack* de ces fils d'exécution ne changeront pas tant qu'ils seront bloqués. La *frame* renvoyée pour un fil d'exécution non bloqué peut ne plus être liée à l'activité courante du fil d'exécution au moment où le code appelant examine la *frame*.

Cette fonction ne devrait être utilisée que pour un usage interne et spécialisé.

sys._debugmallocstats()

Affiche des informations bas-niveau sur la sortie d'erreur à propos de l'état de l'allocateur de mémoire de CPython. Si Python est configuré avec l'option `-with-pydebug`, il effectuera aussi quelques coûteuses vérifications de cohérence interne.

Nouveau dans la version 3.3.

CPython implementation detail : Cette fonction est spécifique à CPython. Le format de sa sortie n'est pas défini ici et pourrait changer.

sys.dllhandle

Nombre entier spécifiant le descripteur de la DLL Python. Disponibilité : Windows.

sys.displayhook (*value*)

Si *value* n'est pas `None`, cette fonction écrit `repr(value)` sur `sys.stdout`, et sauvegarde *value* dans `builtins._`. Si `repr(value)` ne peut pas être encodé avec `sys.stdout.encoding` en utilisant le gestionnaire d'erreur `sys.stdout.errors` (qui est probablement `'strict'`), elle sera encodée avec `sys.stdout.encoding` en utilisant le gestionnaire d'erreur `'backslashreplace'`.

`sys.displayhook` est appelé avec le résultat de l'évaluation d'une *expression* entrée dans une session Python interactive. L'affichage de ces valeurs peut être personnalisé en assignant une autre fonction d'un argument à `sys.displayhook`.

Pseudo-code :

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

Modifié dans la version 3.2 : Utiliser le gestionnaire d'erreur 'backslashreplace' en cas d'`UnicodeEncodeError`.

`sys.dont_write_bytecode`

Si vrai, Python n'essaiera pas d'écrire de fichiers `.pyc` à l'importation de modules source. Cette valeur est initialement définie à `True` ou `False` en fonction de l'option de la ligne de commande `-B` et de la variable d'environnement `PYTHONDONTWRITEBYTECODE`, mais vous pouvez aussi la modifier vous-même pour contrôler la génération des fichiers de *bytecode*.

`sys.excepthook` (*type, value, traceback*)

Cette fonction affiche la *traceback* et l'exception donnée sur `sys.stderr`.

Lorsqu'une exception est levée et n'est pas attrapée, l'interpréteur appelle `sys.excepthook` avec trois arguments, la classe de l'exception, l'instance de l'exception, et un objet *traceback*. Dans une session interactive, cela se produit juste avant que le que l'invite soit rendue. Dans un programme Python, cela se produit juste avant que le programme quitte. La gestion de ces exceptions peut être personnalisé en affectant une autre fonction de trois arguments à `sys.excepthook`.

`sys.__displayhook__`

`sys.__excepthook__`

Ces objets contiennent les valeurs originales de `displayhook` et `excepthook` au début du programme. Elles sont sauvegardées de façon à ce que `displayhook` et `excepthook` puisse être restaurées au cas où elles seraient remplacées par des objets cassés.

`sys.exc_info()`

Cette fonction renvoie un *tuple* de trois valeurs qui donnent des informations sur l'exception actuellement traitée. L'information renvoyée est spécifique à la fois au fil d'exécution courant et à la *stack frame* courante. Si la *stack frame* actuelle ne traite pas d'exception, l'information est extraite de la *stack frame* parente, puis celle appelante, et ainsi de suite jusqu'à trouver une *stack frame* traitant une exception. Ici, « traiter une exception » signifie « exécute une clause *except* ». Pour chaque *stack frame*, seule l'information à propos d'une exception actuellement traitée est accessible.

Si aucune exception n'est actuellement traitée de toute la pile, un *tuple* contenant trois `None` sera renvoyé. Autrement, les valeurs renvoyées sont (*type, value, traceback*). Respectivement *type* reçoit le type de l'exception traitée (une classe fille de `BaseException`), *value* reçoit l'instance de l'exception (une instance du type de l'exception), et *traceback* reçoit un objet *traceback* (voir le Manuel de Référence) qui encapsule la pile d'appels au point où l'exception s'est produite à l'origine.

`sys.exec_prefix`

Une chaîne donnant le préfixe de dossier spécifique au site où les fichiers dépendant de la plateforme sont ins-

tallés. Par défaut, c'est `'/usr/local'`. C'est configurable à la compilation avec l'option `--exec-prefix` du script **configure**. Tous les fichiers de configurations (tel que `pyconfig.h`) sont installés dans le dossier `exec_prefix/lib/pythonX.Y/config`, et les modules sous forme de bibliothèques partagées sont installés dans `exec_prefix/lib/pythonX.Y/lib-dynload`, où `X.Y` est le numéro de version de Python, par exemple 3.2.

Note : Si un *environment virtuel* est actif, cette valeur sera modifiée par `site.py` pour pointer vers l'environnement virtuel. La valeur d'origine sera toujours disponible via `base_exec_prefix`.

`sys.executable`

Une chaîne donnant le chemin absolu vers l'interpréteur Python, un fichier binaire exécutable, sur les système sur lesquels ça a du sens. Si Python n'est pas capable de récupérer le chemin réel de son exécutable, `sys.executable` sera une chaîne vide ou `None`.

`sys.exit([arg])`

Quitte Python. C'est implémenté en levant l'exception `SystemExit`, afin que toutes les actions de nettoyage spécifiées par des clauses *finally* des instructions `try` soient correctement exécutées. Il est aussi possible d'intercepter la tentative de sortie à un niveau au dessus.

L'argument optionnel `arg` peut être un nombre entier donnant l'état de sortie (zéro par défaut), ou un autre type d'objet. Pour les *shells* (et autres), si c'est un entier, zéro signifie « terminé avec succès », et toutes les autres valeurs signifient « terminé anormalement ». La plupart des systèmes imposent qu'il se situe dans la plage 0–127, et leur comportement n'est pas défini pour les autres cas. Certains systèmes peu communs ont pour convention d'assigner un sens particulier à des valeurs spécifiques. Les programmes Unix utilisent généralement 2 pour les erreurs de syntaxe dans les arguments de la ligne de commande, et 1 pour toutes les autres erreurs. Si un autre type est passé, `None` est équivalent à zéro, et tout autre objet est écrit sur `stderr` et donne un code de sortie 1. Typiquement, `sys.exit("some error message")` est un moyen rapide de quitter un programme en cas d'erreur.

Puisque la fonction `exit()` ne fait « que » lever une exception, elle ne fera quitter le processus que si elle est appelée depuis le fil d'exécution principal, et que l'exception n'est pas interceptée.

Modifié dans la version 3.6 : Si une erreur survient lors du nettoyage après que l'interpréteur Python ait intercepté un `SystemExit` (typiquement une erreur en vidant les tampons des sorties standard), le code de sortie est changé à 120.

`sys.flags`

La *struct sequence* `flags` expose l'état des options de ligne de commande. Ces attributs sont en lecture seule.

attribut	option
<code>debug</code>	<code>-d</code>
<code>inspect</code>	<code>-i</code>
<code>interactive</code>	<code>-i</code>
<code>isolated</code>	<code>-I</code>
<code>optimize</code>	<code>-O</code> or <code>-OO</code>
<code>dont_write_bytecode</code>	<code>-B</code>
<code>no_user_site</code>	<code>-s</code>
<code>no_site</code>	<code>-S</code>
<code>ignore_environment</code>	<code>-E</code>
<code>verbose</code>	<code>-v</code>
<code>bytes_warning</code>	<code>-b</code>
<code>quiet</code>	<code>-q</code>
<code>hash_randomization</code>	<code>-R</code>

Modifié dans la version 3.2 : Ajout de l'attribut `quiet` pour la nouvelle option `-q`.

Nouveau dans la version 3.2.3 : L'attribut `hash_randomization`.

Modifié dans la version 3.3 : Suppression de l'attribut obsolète `division_warning`.

Modifié dans la version 3.4 : Added `isolated` attribute for `-I` `isolated` flag.

sys.float_info

Un *struct sequence* contenant des informations à propos du type *float*. Il contient des informations de bas niveau à propos de la précision et de la représentation interne. Les valeurs correspondent aux différentes constantes à propos des nombres à virgule flottantes définies dans le fichier d'entête `float.h`. Voir la section 5.2.4.2.2 de 1999 ISO/IEC C standard [C99], *Characteristics of floating types*, pour plus de détails.

attribut	macro <i>float.h</i>	explication
<code>epsilon</code>	<code>DBL_EPSILON</code>	différence entre 1 et la plus petite valeur plus grande que 1 représentable en <i>float</i>
<code>dig</code>	<code>DBL_DIG</code>	nombre maximum de décimales pouvant être représentées fidèlement dans un <i>float</i> (voir ci-dessous)
<code>mant_dig</code>	<code>DBL_MANT_DIG</code>	précision : nombre de <i>base-radix</i> chiffres dans la mantisse du <i>float</i>
<code>max</code>	<code>DBL_MAX</code>	plus grand <i>float</i> fini représentable
<code>max_exp</code>	<code>DBL_MAX_EXP</code>	plus grand nombre entier <i>e</i> tel que $\text{radix}^{**} (e-1)$ soit représentable sous forme de <i>float</i> finit
<code>max_10_exp</code>	<code>DBL_MAX_10_EXP</code>	plus grand nombre entier <i>e</i> tel que $10^{**}e$ est dans l'intervalle des nombre flottants finis
<code>min</code>	<code>DBL_MIN</code>	plus petit nombre à virgule flottante positif normalisé
<code>min_exp</code>	<code>DBL_MIN_EXP</code>	plus petit entier <i>e</i> tel que $\text{radix}^{**} (e-1)$ est un <i>float</i> normalisé
<code>min_10_exp</code>	<code>DBL_MIN_10_EXP</code>	plus petit nombre entier <i>e</i> tel que $10^{**}e$ est un nombre à virgule flottante normalisé
<code>radix</code>	<code>FLT_RADIX</code>	base de la représentation de l'exposant
<code>rounds</code>	<code>FLT_ROUNDS</code>	constante, nombre entier représentant le mode d'arrondi utilisé pour les opérations arithmétiques. Elle reflète la valeur de la macro système <code>FLT_ROUNDS</code> au moment du démarrage de l'interpréteur. Voir section 5.2.4.4.2.2 de la norme C99 pour une explication des valeurs possibles et de leurs significations.

L'attribut `sys.float_info.dig` nécessite plus d'explications : Si *s* est une chaîne représentant un nombre décimal avec au plus `sys.float_info.dig` chiffres significatifs, alors, convertir *s* en un nombre à virgule flottante puis à nouveau en chaîne redonnera la même valeur :

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'      # decimal string with 15 significant digits
>>> format(float(s), '.15g')    # convert to float and back -> same value
'3.14159265358979'
```

Cependant, pour les chaînes avec plus de `sys.float_info.dig` chiffres significatifs, ce n'est pas toujours vrai :

```
>>> s = '9876543211234567'     # 16 significant digits is too many!
>>> format(float(s), '.16g')    # conversion changes value
'9876543211234568'
```

sys.float_repr_style

Une chaîne indiquant comment la fonction `repr()` se comporte avec les nombres à virgule flottante. Si la chaîne a la valeur `'short'`, alors pour un *float* finit *x*, `repr(x)` essaye de donner une courte chaîne tel que `float(repr(x)) == x`. C'est le comportement typique à partir de Python 3.1. Autrement, `float_repr_style` a la valeur `'legacy'` et `repr(x)` se comporte comme les versions antérieures à 3.1. Nouveau dans la version 3.1.

sys.getallocatedblocks()

Renvoie le nombre de blocs mémoire actuellement alloués par l'interpréteur, peu importe leur taille. Cette fonction

est principalement utile pour pister les fuites de mémoire. À cause des caches internes de l'interpréteur, le résultat peut varier d'un appel à l'autre. Appeler `_clear_type_cache()` et `gc.collect()` peut permettre d'obtenir des résultats plus prévisibles.

Si Python n'arrive pas à calculer raisonnablement cette information, `getallocatedblocks()` est autorisé à renvoyer 0 à la place.

Nouveau dans la version 3.4.

`sys.getcheckinterval()`

Renvoie le *check interval* de l'interpréteur, voir `setcheckinterval()`.

Obsolète depuis la version 3.2 : Utilisez plutôt `getswitchinterval()`.

`sys.getdefaultencoding()`

Renvoie le nom du codage par défaut actuellement utilisé par l'implémentation *Unicode* pour coder les chaînes.

`sys.getdlopenflags()`

Renvoie la valeur actuelle des *flags* utilisés par les appels de `dlopen()`. Les noms symboliques valeurs peuvent être trouvées dans le module `os`. (Ce sont les constantes `RTLD_XXX` e.g. `os.RTLD_LAZY`). Disponibilité : Unix.

`sys.getfilesystemencoding()`

Donne le nom de l'encodage utilisé pour les conversions entre les noms de fichiers Unicode et les noms de fichiers en octets. Pour une compatibilité optimale, les noms de fichiers devraient toujours être représentés sous forme de chaînes de caractères, cependant les représenter sous forme d'objet *bytes* est aussi accepté. Les fonctions acceptant ou renvoyant des noms de fichiers devraient supporter les deux (*str* ou *bytes*), et convertir en interne dans la représentation du système.

Cet encodage est toujours compatible avec ASCII.

Les fonctions `os.fsencode()` et `os.fsdecode()` devraient être utilisées pour s'assurer qu'un encodage et un gestionnaire d'erreurs correct sont utilisés.

— Sur Mac OS X, l'encodage est 'utf-8'.

— Sur Unix, l'encodage est celui des paramètres régionaux.

— Sur Windows, l'encodage peut être 'utf-8' ou 'mbcs', en fonction des paramètres de l'utilisateur.

Modifié dans la version 3.2 : `getfilesystemencoding()` ne peut plus renvoyer `None`.

Modifié dans la version 3.6 : Sur Windows, on est plus assuré d'obtenir 'mbcs'. Voir la [PEP 529](#) et `_enablelegacywindowsfsencoding()` pour plus d'informations.

`sys.getfilesystemencodeerrors()`

Donne le nom du mode de gestion d'erreur utilisé lors de la conversion des noms de fichiers entre Unicode et octets. Le nom de l'encodage est renvoyé par `getfilesystemencoding()`.

Les fonctions `os.fsencode()` et `os.fsdecode()` devraient être utilisées pour s'assurer qu'un encodage et un gestionnaire d'erreurs correct sont utilisés.

Nouveau dans la version 3.6.

`sys.getrefcount(object)`

Donne le nombre de référence de l'objet *object*. Le nombre renvoyé est généralement d'une référence de plus qu'attendu, puisqu'il compte la référence (temporaire) de l'argument à `getrefcount()`.

`sys.getrecursionlimit()`

Donne la limite actuelle de la limite de récursion, la profondeur maximum de la pile de l'interpréteur. Cette limite empêche Python de planter lors d'une récursion infinie à cause d'un débordement de la pile. Elle peut être modifiée par `setrecursionlimit()`.

`sys.getsizeof(object[, default])`

Donne la taille d'un objet en octets. L'objet peut être de n'importe quel type. Le résultat sera correct pour tous les objets natifs, mais le résultat peut ne pas être toujours vrai pour les extensions, la valeur étant dépendante de l'implémentation.

Seule la mémoire directement attribuée à l'objet est prise en compte, pas la mémoire consommée par les objets vers lesquels il a des références.

S'il est fourni, *default* sera renvoyé si l'objet ne fournit aucun moyen de récupérer sa taille. Sinon, une exception `TypeError` sera levée.

`getsizeof()` appelle la méthode `__sizeof__` de l'objet, et s'il est géré par lui, ajoute le surcoût du ramasse-miettes.

Voir la [recursive sizeof recipe](#) pour un exemple d'utilisation récursive de `getsizeof()` pour trouver la taille d'un contenant et de son contenu.

`sys.getswitchinterval()`

Renvoie la valeur du *thread switch interval* de l'interpréteur, voir `setswitchinterval()`.

Nouveau dans la version 3.2.

`sys._getframe([depth])`

Renvoie une *frame* de la pile d'appels. Si le nombre entier optionnel *depth* est donné, la *frame* donnée sera de *depth* appels depuis le haut de la pile. Si c'est plus profond que la hauteur de la pile, une exception `ValueError` est levée. La profondeur par défaut est zéro, donnant ainsi la *frame* du dessus de la pile.

CPython implementation detail : Cette fonction ne devrait être utilisée que pour une utilisation interne et spécifique. Il n'est pas garanti qu'elle existe dans toutes les implémentations de Python.

`sys.getprofile()`

Renvoie la fonction de profilage tel que défini par `setprofile()`.

`sys.gettrace()`

Renvoie la fonction de traçage tel que définie par `settrace()`.

CPython implementation detail : La fonction `gettrace()` ne sert que pour implémenter des débogueurs, des *profilers*, outils d'analyse de couverture, etc.... Son comportement dépend de l'implémentation et non du langage, elle n'est donc pas forcément disponible dans toutes les implémentations de Python.

`sys.getwindowsversion()`

Renvoie un tuple nommé décrivant la version de Windows en cours d'exécution. Les attributs nommés sont *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* et *platform_version*. *service_pack* contient une string, *platform_version* un tuple de trois valeurs, et tous les autres sont des nombres entiers. Ces attributs sont également accessibles par leur nom, donc `sys.getwindowsversion()[0]` est équivalent à `sys.getwindowsversion().major`. Pour des raisons de compatibilité avec les versions antérieures, seuls les 5 premiers éléments sont accessibles par leur indice.

platform sera 2 (VER_PLATFORM_WIN32_NT).

product_type peut être une des valeurs suivantes :

Constante	Signification
1 (VER_NT_WORKSTATION)	Le système une station de travail.
2 (VER_NT_DOMAIN_CONTROLLER)	Le système est un contrôleur de domaine.
3 (VER_NT_SERVER)	Le système est un serveur, mais pas un contrôleur de domaine.

Cette fonction enveloppe la fonction Win32 `GetVersionEx()`. Voir la documentation de Microsoft sur `OSVERSIONINFOEX()` pour plus d'informations sur ces champs.

platform_version donne précisément la version majeure, mineure, et numéro de compilation du système d'exploitation sous-jacent, plutôt que la version émulée pour ce processus. Il est destiné à être utilisé pour de la journalisation plutôt que pour la détection de fonctionnalités.

Disponibilité : Windows.

Modifié dans la version 3.2 : Changé en un tuple nommé, et ajout de *service_pack_minor*, *service_pack_major*, *suite_mask*, et *product_type*.

Modifié dans la version 3.6 : Ajout de *platform_version*

`sys.get_asyncgen_hooks()`

Renvoie un objet *asyncgen_hooks*, qui est semblable à un *namedtuple* de la forme (*firstiter*, *finalizer*), où *firstiter* et *finalizer* sont soit `None` ou des fonctions qui prennent un *asynchronous generator iterator* comme argument, et sont utilisées pour planifier la finalisation d'un générateur asynchrone par un *event loop*.

Nouveau dans la version 3.6 : Voir la [PEP 525](#) pour plus d'informations.

Note : Cette fonction a été ajoutée à titre provisoire (voir la [PEP 411](#) pour plus d'informations.)

`sys.get_coroutine_wrapper()`

Renvoie `None`, ou un *wrapper* donné via `set_coroutine_wrapper()`.

Nouveau dans la version 3.5 : Voir la [PEP 492](#) pour plus d'informations.

Note : Cette fonction a été ajoutée à titre provisoire (Voir la [PEP 411](#) pour plus d'informations.) Utilisez la uniquement à des fins de débogage.

`sys.hash_info`

Une *struct sequence* donnant les paramètres de l'implémentation de la fonction de hachage de nombres. Pour plus d'informations sur le hachage des types numériques, consultez *Hachage des types numériques*.

attribut	explication
<code>width</code>	Nombre de bits des valeurs de <i>hash</i>
<code>modulus</code>	contient le premier <i>P</i> utilisé dans le modulo pour les <i>hash</i> numériques
<code>inf</code>	valeur du <i>hash</i> pour un infini positif
<code>nan</code>	valeur du <i>hash</i> pour un <i>nan</i>
<code>imag</code>	multiplicateur utilisé pour la partie imaginaire d'un nombre complexe
<code>algorithm</code>	nom de l'algorithme pour le hachage des <i>str</i> , <i>bytes</i> , et <i>memoryview</i>
<code>hash_bits</code>	taille de la sortie interne de l'algorithme de hachage
<code>seed_bits</code>	taille de la <i>seed key</i> utilisée par l'algorithme de hachage

Nouveau dans la version 3.2.

Modifié dans la version 3.4 : Ajout de *algorithm*, *hash_bits* et *seed_bits*

`sys.hexversion`

Le numéro de version codé sous forme d'un seul nombre entier. Ce numéro augmente avec chaque version, y compris pour les versions hors production. Par exemple, pour vérifier que l'interpréteur Python est au moins la version 1.5, utilisez :

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

Cet attribut s'appelle *hexversion* dans le sens où il ne semble avoir du sens que s'il est regardé après avoir été passé à la fonction native `hex()`. La *struct sequence* `sys.version_info` représente la même information d'une manière plus humaine.

Consultez `apiabiversion` pour plus d'informations sur *hexversion*.

`sys.implementation`

Un objet contenant des informations sur l'implémentation de la version actuelle de l'interpréteur Python. Les attributs suivants existent obligatoirement sur toutes les implémentations Python.

name est l'identifiant de l'implémentation, e.g. 'cpython'. Cette chaîne est définie par l'implémentation de Python, mais sera toujours en minuscule.

version est un *named tuple*, du même format que `sys.version_info`. Il représente la version de l'**implémentation** de Python. C'est une information différente de la version du **langage** auquel l'interpréteur actuel se conforme (donnée par `sys.version_info`). Par exemple, pour PyPy 1.8 `sys.implementation.version` peut valoir `sys.version_info(1, 8, 0, 'final', 0)`, alors que `sys.version_info`

peut valoir `sys.version_info(2, 7, 2, 'final', 0)`. Pour CPython ces deux valeurs sont identiques puisque c'est l'implémentation de référence.

`hexversion` est la version de l'implémentation sous forme hexadécimale, comme `sys.hexversion`.

`cache_tag` est la balise utilisée par le mécanisme d'importation dans les noms de fichiers des modules mis en cache. Par convention, il devrait se composer du nom et de la version de l'implémentation, comme ``cpython-33'``. Cependant, une implémentation Python peut utiliser une autre valeur si nécessaire. ``cache_tag`` à `None` signifie que la mise en cache des modules doit être désactivée.

`sys.implementation` peut contenir d'autres attributs spécifiques à l'implémentation de Python. Ces attributs spécifiques doivent commencer par un *underscore*, et ne sont pas documentés ici. Indépendamment de son contenu, `sys.implementation` ne change jamais durant l'exécution de l'interpréteur, ni entre les versions d'une même implémentation. (Il peut cependant changer entre les versions du langage Python.) Voir la [PEP 421](#) pour plus d'informations.

Nouveau dans la version 3.3.

`sys.int_info`

Un *struct sequence* qui contient des informations sur la représentation interne des entiers de Python. Les attributs sont en lecture seule.

Attribut	Explication
<code>bits_per_digit</code>	nombre de bits utilisés pour chaque chiffre. Les entiers Python sont stockés en interne en base $2^{**int_info.bits_per_digit}$
<code>sizeof_digit</code>	taille en octets du type C utilisé pour représenter un chiffre

Nouveau dans la version 3.1.

`sys.__interactivehook__`

Lorsque cet attribut existe, sa valeur est automatiquement appelée (sans argument) par l'interpréteur lors de son démarrage en mode interactif. L'appel se fait après que le fichier `PYTHONSTARTUP` soit lu, afin que vous puissiez y configurer votre fonction. *Configuré* par le module `site`.

Nouveau dans la version 3.4.

`sys.intern(string)`

Ajoute *string* dans le tableau des chaînes « internées » et renvoie la chaîne internée – qui peut être *string* elle-même ou une copie. Interner une chaîne de caractères permet de gagner un peu de performance lors de l'accès aux dictionnaires – si les clés du dictionnaire et la clé recherchée sont internées, les comparaisons de clés (après le hachage) pourront se faire en comparant les pointeurs plutôt que caractère par caractère. Normalement, les noms utilisés dans les programmes Python sont automatiquement internés, et les dictionnaires utilisés pour stocker les attributs de modules, de classes, ou d'instances ont aussi leurs clés internées.

Les chaînes internées ne sont pas immortelles ; vous devez garder une référence à la valeur renvoyée par `intern()` pour en bénéficier.

`sys.is_finalizing()`

Donne *True* si l'interpréteur Python est *en train de s'arrêter*, et *False* dans le cas contraire.

Nouveau dans la version 3.5.

`sys.last_type`

`sys.last_value`

`sys.last_traceback`

Ces trois variables ne sont pas toujours définies. Elles sont définies lorsqu'une exception n'est pas gérée et que l'interpréteur affiche un message d'erreur et une *stacktrace*. Elles sont là pour permettre à un utilisateur, en mode interactif, d'importer un module de débogage et de faire son débogage post-mortem sans avoir à ré-exécuter la commande qui a causé l'erreur. (L'utilisation typique pour entrer dans le débogueur post-mortem est `import pdb; pdb.pm()`, voir [pdb](#) pour plus d'informations.).

La signification de ces variables est la même que celle des valeurs renvoyées par `exc_info()` ci-dessus.

sys.maxsize

Un entier donnant à la valeur maximale qu'une variable de type `Py_ssize_t` peut prendre. C'est typiquement $2^{*31} - 1$ sur une plateforme 32 bits et $2^{*63} - 1$ sur une plateforme 64 bits.

sys.maxunicode

Un entier donnant la valeur du plus grand point de code Unicode, c'est-à-dire 1114111 (`0x10FFFF` en hexa-décimal).

Modifié dans la version 3.3 : Avant la [PEP 393](#), `sys.maxunicode` valait soit `0xFFFF` soit `0x10FFFF`, en fonction l'option de configuration qui spécifiait si les caractères Unicode étaient stockés en UCS-2 ou UCS-4.

sys.meta_path

Une liste d'objets *meta path finder* qui ont leur méthode `find_spec()` appelée pour voir si un des objets peut trouver le module à importer. La méthode `find_spec()` est appelée avec au moins le nom absolu du module importé. Si le module à importer est contenu dans un paquet, l'attribut `__path__` du paquet parent est donné en deuxième argument. La méthode renvoie un *module spec*, ou `None` si le module ne peut être trouvé.

Voir aussi :

[`importlib.abc.MetaPathFinder`](#) La classe de base abstraite définissant l'interface des objets *finder* de *meta_path*.

[`importlib.machinery.ModuleSpec`](#) La classe concrète dont `find_spec()` devrait renvoyer des instances.

Modifié dans la version 3.4 : Les *Module specs* ont été introduits en Python 3.4, par la [PEP 451](#). Les versions antérieures de Python cherchaient une méthode appelée `find_module()`. Celle-ci est toujours appelée en dernier recours, dans le cas où une *meta_path* n'a pas de méthode `find_spec()`.

sys.modules

Un dictionnaire faisant correspondre des noms de modules à des modules déjà chargés. Il peut être manipulé, entre autre, pour forcer un module à être rechargé. Cependant, le remplacer ne fonctionnera pas forcément comme prévu et en supprimer des éléments essentiels peut planter Python.

sys.path

Une liste de chaînes de caractères spécifiant les chemins de recherche des modules, initialisée à partir de la variable d'environnement `PYTHONPATH` et d'une valeur par défaut dépendante de l'installation.

Puisqu'il est initialisé au démarrage du programme, le premier élément de cette liste, `path[0]`, est le dossier contenant le script qui a été utilisé pour invoquer l'interpréteur Python. Si le dossier du script n'est pas disponible (typiquement, si l'interpréteur est invoqué interactivement ou si le script est lu à partir d'une entrée standard), `path[0]` sera une chaîne vide, qui indiquera à Python de chercher des modules dans le dossier actuel. Notez que le dossier du script est inséré *avant* les dossiers de `PYTHONPATH`.

Un programme est libre de modifier cette liste pour ses propres besoins. Seuls des *str* ou des *bytes* ne devraient être ajoutés à `sys.path`, tous les autres types de données étant ignorés durant l'importation.

Voir aussi :

Le module [`site`](#) décrit comment utiliser les fichiers *.pth* pour étendre `sys.path`.

sys.path_hooks

Une liste d'appelables d'un argument, *path*, pour essayer de créer un *finder* pour ce chemin. Si un *finder* peut être créé, il doit être renvoyé par l'appelable, sinon une *ImportError* doit être levée.

Précisé à l'origine dans la [PEP 302](#).

sys.path_importer_cache

Un dictionnaire faisant office de cache pour les objets *finder*. Les clés sont les chemins qui ont été passés à `sys.path_hooks` et les valeurs sont les *finders* trouvés. Si un chemin est valide selon le système de fichiers mais qu'aucun *finder* n'est trouvé dans `sys.path_hooks`, `None` est stocké.

Précisé à l'origine dans la [PEP 302](#).

Modifié dans la version 3.3 : `None` est stocké à la place de [`imp.NullImporter`](#) si aucun localisateur n'est trouvé.

sys.platform

Cette chaîne contient un identificateur de plateforme qui peut être typiquement utilisé pour ajouter des composants spécifiques à `sys.path`.

Pour les systèmes Unix, sauf sur Linux, c'est le nom de l'OS en minuscules comme renvoyé par `uname -s` suivi de la première partie de la version comme renvoyée par `uname -r`, e.g. 'sunos5' ou 'freebsd8', *au moment où Python a été compilé*. A moins que vous ne souhaitiez tester pour une version spécifique du système, vous pouvez faire comme suit :

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

Pour les autres systèmes, les valeurs sont :

Le système une station de travail.	Valeur pour <code>platform</code>
Linux	'linux'
Windows	'win32'
Windows/Cygwin	'cygwin'
Mac OS X	'darwin'

Modifié dans la version 3.3 : Sur Linux, `sys.platform` ne contient plus la version majeure, c'est toujours 'linux', au lieu de 'linux2' ou 'linux3'. Comme les anciennes versions de Python incluent le numéro de version, il est recommandé de toujours utiliser `startswith`, tel qu'utilisé ci-dessus.

Voir aussi :

`os.name` a une granularité plus grossière. `os.uname()` donne des informations sur la version dépendantes du système.

Le module `platform` fournit des vérifications détaillées pour l'identité du système.

sys.prefix

Une chaîne donnant le préfixe de répertoire spécifique au site dans lequel les fichiers Python indépendants de la plate-forme sont installés. Par défaut, c'est '/usr/local'. Ceci peut être défini à la compilation en passant l'argument `--prefix` au script `configure`. La collection principale des modules de la bibliothèque Python est installée dans le dossier `prefix/lib/pythonX.Y` et les entêtes indépendantes de la plateforme (toutes sauf `pyconfig.h`) sont stockées dans `prefix/include/pythonX.Y`, où `X.Y` est le numéro de version de Python, par exemple 3.2.

Note : Si a *environnement virtuel* est activé, cette valeur sera changée par `site.py` pour pointer vers l'environnement virtuel. La valeur donnée au moment de la compilation de Python sera toujours disponible, dans `base_prefix`.

sys.ps1**sys.ps2**

Chaînes spécifiant l'invite primaire et secondaire de l'interpréteur. Celles-ci ne sont définies que si l'interpréteur est en mode interactif. Dans ce cas, leurs valeurs initiales sont '>>>' et '...'. Si un objet qui n'est pas une chaîne est assigné à l'une ou l'autre variable, sa méthode `str()` sera appelée à chaque fois que l'interpréteur se prépare à lire une nouvelle commande interactive, c'est donc utilisable pour implémenter une invite dynamique.

sys.setcheckinterval(interval)

Définit l'« intervalle de vérification » de l'interpréteur. Ce nombre entier détermine la fréquence à laquelle l'interpréteur effectue des tâches périodiques tels que la commutation de fil d'exécution et la gestion de signaux. La valeur par défaut est 100, ce qui signifie que le contrôle est effectué toutes les 100 instructions virtuelles Python. L'augmenter peut améliorer les performances des programmes utilisant des fils d'exécution. Le paramétrer à une valeur inférieure ou égale à zéro permet d'effectuer ces tâches à chaque instruction virtuelle, maximisant ainsi la réactivité mais aussi son surcoût.

Obsolète depuis la version 3.2 : Cette fonction n'a plus aucun effet : La logique interne de commutation de fils d'exécution et de gestion des tâches asynchrones ayant été réécrite. Utilisez `setswitchinterval()` à la place.

`sys.setdlopenflags(n)`

Définit les options utilisées par l'interpréteur lors des appels à `dlopen()`, typiquement utilisé par l'interpréteur pour charger des modules d'extension. Permet entre autre de résoudre tardivement les symboles lors des importations de modules (si appelé `sys.setdlopenflags(0)`). Pour partager les symboles entre modules, appelez `sys.setdlopenflags(os.RTLD_GLOBAL)`. Les noms pour les valeurs de ces options peuvent être trouvés dans le module `os` (ce sont les constantes `RTLD_XXX`, comme `os.RTLD_LAZY`).

Disponibilité : Unix.

`sys.setprofile(profilefunc)`

Définit la fonction de profilage du système, qui vous permet d'implémenter un profileur de code source Python en Python. Voir le chapitre *The Python Profilers* pour plus d'informations sur le profileur Python. La fonction de profilage du système est appelée de la même façon que la fonction `trace` du (voir `settrace()`), mais elle est appelée pour des événements différents, par exemple elle n'est pas appelée à chaque ligne de code exécutée (seulement sur appel et retours, mais l'événement pour les retours est appelé même en cas d'exception). Cette fonction est locale au fil d'exécution, et il n'existe aucun moyen, du point de vue du profileur, de prendre conscience des changements de contextes entre fils d'exécution, ça n'a donc aucun sens d'utiliser cette fonction dans un contexte *multithread*. Sa valeur de retour n'est pas utilisée, elle peut simplement renvoyer `None`.

Les fonctions de traçage doivent avoir trois arguments : *frame*, *event*, et *arg*. *frame* est la *stack frame* actuelle. *event* est une chaîne de caractères pouvant valoir : `'call'`, `'return'`, `'c_call'`, `'c_return'` ou `'c_exception'`. *arg* dépend du type de l'évènement.

Les événements ont la signification suivante :

'call' Une fonction est appelée (ou Python entre dans un autre bloc de code). La fonction de traçage est appelée, *arg* est `None`.

'return' La fonction (ou un autre type de bloc) est sur le point de se terminer. La fonction de traçage est appelée, *arg* est la valeur qui sera renvoyée, ou `None` si l'évènement est causé par la levée d'une exception.

'c_call' Une fonction C est sur le point d'être appelée. C'est soit une fonction d'extension ou une fonction native. *arg* représente la fonction C.

'c_return' Une fonction C a renvoyé une valeur. *arg* représente la fonction C.

'c_exception' Une fonction C a levé une exception. *arg* représente la fonction C.

`sys.setrecursionlimit(limit)`

Définit la profondeur maximale de la pile de l'interpréteur Python à *limit*. Cette limite empêche une récursion infinie de provoquer un débordement de la pile C et ainsi un crash de Python.

La limite haute dépend de la plate-forme. Un utilisateur pourrait avoir besoin de remonter la limite, lorsque son programme nécessite une récursion profonde, si sa plate-forme le permet. Cela doit être fait avec précaution, car une limite trop élevée peut conduire à un crash.

Si la nouvelle limite est plus basse que la profondeur actuelle, une `RecursionError` est levée.

Modifié dans la version 3.5.1 : Une `RecursionError` est maintenant levée si la nouvelle limite est plus basse que la profondeur de récursion actuelle.

`sys.setswitchinterval(interval)`

Configure l'intervalle de bascule des fils d'exécution de l'interpréteur (en secondes). Ce nombre à virgule flottante détermine la durée idéale allouée aux fils d'exécution en cours d'exécution (durée appelée *timeslices*). Notez que la durée observée peut être plus grande, typiquement si des fonctions ou méthodes prenant beaucoup de temps sont utilisées. Aussi, le choix du fil d'exécution prenant la main à la fin de l'intervalle revient au système d'exploitation. L'interpréteur n'a pas son propre ordonnanceur.

Nouveau dans la version 3.2.

`sys.settrace(tracefunc)`

Définit la fonction de traçage du système, qui vous permet d'implémenter un débogueur de code source Python en Python. Cette fonction est locale au fil d'exécution courant. Pour qu'un débogueur puisse gérer plusieurs fils

d'exécution, il doit enregistrer sa fonction en appelant `settrace()` pour chaque fil d'exécution qu'il souhaite surveiller.

Les fonctions de traçage doivent avoir trois arguments : *frame*, *event*, et *arg*. *frame* est la *stack frame* actuelle. *event* est une chaîne de caractères pouvant valoir : 'call', 'line', 'return', 'exception'. *arg* dépend du type de l'évènement.

La fonction de traçage est appelée (avec *event* à 'call') à chaque fois que l'interpréteur entre dans un nouveau *scope*. Elle doit renvoyer une référence à une fonction de traçage locale à utiliser pour ce *scope*, ou `None` si le *Scope* ne doit pas être tracé.

La fonction de traçage doit renvoyer une référence à elle-même (ou à une autre fonction de traçage pour un traçage ultérieur dans cette portée), ou `None` pour désactiver le traçage dans cette portée.

Les événements ont la signification suivante :

'call' Une fonction est appelée (un un bloc de code). La fonction de traçage globale est appelée, *arg* est `None`, la valeur renvoyée donne la fonction de traçage locale.

'line' L'interpréteur est sur le point d'exécuter une nouvelle ligne de code ou de ré-exécuter la condition d'une boucle. La fonction de traçage locale est appelée, *arg* vaut `None`, et la valeur de retour donne la nouvelle fonction de traçage locale. Voir `Objects/lnotab_notes.txt` pour une explication détaillée de ce mécanisme.

'return' La fonction (ou un autre type de bloc) est sur le point de se terminer. La fonction de traçage locale est appelée, *arg* est la valeur qui sera renvoyée, ou `None` si l'évènement est causé par la levée d'une exception. La valeur renvoyée par la fonction de traçage est ignorée.

'exception' Une exception est survenue. La fonction de traçage locale est appelée, *arg* est le *tuple* (exception, valeur, traceback), la valeur renvoyée spécifie la nouvelle fonction de traçage locale.

Remarquez que, comme une exception se propage au travers de toute chaîne d'appelants, un événement 'exception' est généré à chaque niveau.

Pour plus d'informations sur les objets code et objets représentant une *frame* de la pile, consultez `types`.

CPython implementation detail : La fonction `settrace()` est destinée uniquement à l'implémentation de débogueurs, de profileurs, d'outils d'analyse de couverture et d'autres outils similaires. Son comportement fait partie de l'implémentation, plutôt que de la définition du langage, et peut donc ne pas être disponible dans toutes les implémentations de Python.

`sys.set_asyncgen_hooks` (*firstiter*, *finalizer*)

Accepte deux arguments optionnels nommés, qui sont appelables qui acceptent un *asynchronous generator iterator* comme argument. L'appelable *firstiter* sera appelé lorsqu'un générateur asynchrone sera itéré pour la première fois, et l'appelable *finalizer* sera appelé lorsqu'un générateur asynchrone est sur le point d'être détruit.

Nouveau dans la version 3.6 : Voir la [PEP 525](#) pour plus de détails. Pour un exemple de *finalizer*, voir l'implémentation de `asyncio.Loop.shutdown_asyncgens` dans `Lib/asyncio/base_events.py`

Note : Cette fonction à été ajoutée à titre provisoire (voir la [PEP 411](#) pour plus d'informations.)

`sys.set_coroutine_wrapper` (*wrapper*)

Permet d'intercepter la création de *coroutine* (uniquement celles créés via `async def`, les générateurs décorés par `types.coroutine()` ou `asyncio.coroutine()` ne seront pas interceptés).

L'argument *wrapper* doit être soit :

- un callable qui accepte un argument (une coroutine);
- `None`, pour réinitialiser le *wrapper*.

S'il est appelé deux fois, le nouveau *wrapper* remplace le précédent.

L'appelable *wrapper* ne peut pas définir de nouvelles coroutines, ni directement, ni indirectement :

```
def wrapper(coro):
    async def wrap(coro):
        return await coro
```

(suite sur la page suivante)

(suite de la page précédente)

```

    return wrap(coro)
sys.set_coroutine_wrapper(wrapper)

async def foo():
    pass

# The following line will fail with a RuntimeError, because
# ``wrapper`` creates a ``wrap(coro)`` coroutine:
foo()

```

Voir aussi `get_coroutine_wrapper()`.

Nouveau dans la version 3.5 : Voir la [PEP 492](#) pour plus d'informations.

Note : Cette fonction a été ajoutée à titre provisoire (Voir la [PEP 411](#) pour plus d'informations.) Utilisez la uniquement à des fins de débogage.

`sys._enablelegacywindowsfsencoding()`

Change l'encodage et le mode de gestion d'erreur par défaut du système de fichiers à *mbcs* et *replace* respectivement, par cohérence avec les versions de Python antérieures à la 3.6.

Équivaut à définir la variable d'environnement `PYTHONLEGACYWINDOWSFSENCODING` avant de lancer Python.

Disponibilité : Windows

Nouveau dans la version 3.6 : Voir la [PEP 529](#) pour plus d'informations.

`sys.stdin`

`sys.stdout`

`sys.stderr`

objets fichiers utilisés par l'interpréteur pour l'entrée standard, la sortie standard et la sortie d'erreurs :

- `stdin` est utilisé pour toutes les entrées interactives (y compris les appels à `input()`)
- `stdout` est utilisé pour la sortie de `print()`, des *expression* et pour les invites de `input()` ;
- Les invites de l'interpréteur et ses messages d'erreur sont écrits sur `stderr`.

Ces flux sont de classiques *fichiers texte* comme ceux renvoyés par la fonction `open()`. Leurs paramètres sont choisis comme suit :

- L'encodage des caractères dépend de la plate-forme. Sous Windows, si le flux est interactif (c'est-à-dire si sa méthode `isatty()` donne `True`), l'encodage de la console est utilisée, sinon un encodage Windows. Sous d'autres plateformes, l'encodage local est utilisé (voir `locale.getpreferredencoding()`). Sous toutes les plates-formes cependant, vous pouvez remplacer cette valeur en définissant la variable d'environnement `PYTHONIOENCODING` avant de démarrer Python.
- En mode interactif, les entrées et sorties standards passent par un tampon d'une ligne. Autrement, elles passent par blocs dans un tampon, comme les fichiers textes classiques. Vous pouvez. Vous pouvez remplacer cette valeur avec l'option `-u` en ligne de commande.

Note : Pour écrire ou lire des données binaires depuis ou vers les flux standards, utilisez l'objet sous-jacent *buffer*. Par exemple, pour écrire des octets sur `stdout`, utilisez `sys.stdout.buffer.write(b'abc')`.

Cependant, si vous écrivez une bibliothèque (ou ne contrôlez pas dans quel contexte son code sera exécuté), sachez que les flux standards peuvent être remplacés par des objets de type fichier tel un `io.StringIO` qui n'ont pas l'attribut `buffer`.

`sys.__stdin__`

`sys.__stdout__`

`sys.__stderr__`

Ces objets contiennent les valeurs d'origine de `stdin`, `stderr` et `stdout` tel que présentes au début du pro-

gramme. Ils sont utilisés pendant la finalisation, et peuvent être utiles pour écrire dans le vrai flux standard, peu importe si l'objet `sys.std*` a été redirigé.

Ils peuvent également être utilisés pour restaurer les entrées / sorties d'origine, au cas où ils auraient été écrasés par des objets cassés, cependant la bonne façon de faire serait de sauvegarder explicitement les flux avant de les remplacer et ainsi pouvoir les restaurer.

Note : Dans certaines cas, `stdin`, `stdout` et `stderr` ainsi que les valeurs initiales `__stdin__`, `__stdout__` et `__stderr__` peuvent être `None`. C'est typiquement le cas pour les applications graphiques sur Windows qui ne sont pas connectées à une console, ou les applications Python démarrées avec **pythonw**.

`sys.thread_info`

Une *struct sequence* contenant des informations sur l'implémentation des fils d'exécution.

Attribut	Explication
<code>name</code>	Nom de l'implémentation des fils d'exécution : — <code>'nt'</code> : Fils d'exécution Windows — <code>'pthread'</code> : Fils d'exécution POSIX — <code>'solaris'</code> : Fils d'exécution Solaris
<code>lock</code>	Nom de l'implémentation du système de verrou : — <code>'semaphore'</code> : Verrou utilisant une sémaphore — <code>'mutex+cond'</code> : Un verrou utilisant un <i>mutex</i> et une <i>condition variable</i> — <code>None</code> si cette information n'est pas connue
<code>version</code>	Nom et version de l'implémentation des fils d'exécution, c'est une chaîne, ou <code>None</code> si ces informations sont inconnues.

Nouveau dans la version 3.3.

`sys.tracebacklimit`

Lorsque cette variable contient un nombre entier, elle détermine la profondeur maximum de la pile d'appels affichée lorsqu'une exception non gérée se produit. La valeur par défaut est 1000, lorsque cette valeur est égale ou inférieure à 0, la pile d'appels n'est pas affichée, seul le type et la valeur de l'exception sont le sont.

`sys.version`

Une chaîne contenant le numéro de version de l'interpréteur Python, ainsi que d'autres informations comme le numéro de compilation et le compilateur utilisé. Cette chaîne est affichée lorsque l'interpréteur est démarré en mode interactif. N'essayez pas d'en extraire des informations de version, utilisez plutôt *version_info* et les fonctions fournies par le module *platform*.

`sys.api_version`

La version de l'API C pour cet interpréteur. Les développeurs peuvent trouver cette information utile en déboguant des conflits de versions entre Python et des modules d'extension.

`sys.version_info`

Un *tuple* contenant les cinq composants du numéro de version : *major*, *minor*, *micro*, *releaselevel* et *serial*. Toutes les valeurs sauf *releaselevel* sont des nombres entiers. *releaselevel* peut valoir `'alpha'`, `'beta'`, `'candidate'`, ou `'final'`. La valeur de *version_info* pour Python 2.0 est (2, 0, 0, 'final', 0). Ces attributs sont aussi accessibles par leur nom, ainsi `sys.version_info[0]` est équivalent à `sys.version_info.major`, et ainsi de suite.

Modifié dans la version 3.1 : Ajout des attributs nommés.

`sys.warnoptions`

C'est une spécificité de l'implémentation de la gestion des avertissements. Ne modifiez pas cette valeur. Reportez-vous au module *warnings* pour plus d'informations sur le gestionnaire d'avertissements.

`sys.winver`

Le numéro de version utilisé pour construire les clefs de registre sous Windows. Elle est stockée en tant que *string resource* 1000 dans la DLL Python. Cette valeur équivaut typiquement aux trois premiers caractères de *version*. Elle est fournie par le module *sys* à titre d'information, et la modifier n'a aucun effet sur les clés de registre utilisées par Python. Disponibilité : Windows.

`sys._xoptions`

Un dictionnaire des différentes options spécifiques à l'implémentation passées en ligne de commande via l'option `-X`. Aux noms des options correspondent soit leur valeur, si elle est donnée explicitement, soit à *True*. Exemple :

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython implementation detail : C'est un moyen spécifique à CPython pour accéder aux options passées via l'option `-X`. D'autres implémentations pourraient les exposer par d'autres moyens, ou pas du tout. Nouveau dans la version 3.2.

Citations

29.2 `sysconfig` — Provide access to Python's configuration information

Nouveau dans la version 3.2.

Source code : [Lib/sysconfig.py](#)

The *sysconfig* module provides access to Python's configuration information like the list of installation paths and the configuration variables relevant for the current platform.

29.2.1 Configuration variables

A Python distribution contains a `Makefile` and a `pyconfig.h` header file that are necessary to build both the Python binary itself and third-party C extensions compiled using *distutils*.

sysconfig puts all variables found in these files in a dictionary that can be accessed using `get_config_vars()` or `get_config_var()`.

Notice that on Windows, it's a much smaller set.

`sysconfig.get_config_vars(*args)`

With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return `None`.

`sysconfig.get_config_var(name)`

Return the value of a single variable *name*. Equivalent to `get_config_vars().get(name)`.

If *name* is not found, return `None`.

Example of usage :

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

29.2.2 Installation paths

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in `sysconfig` under unique identifiers based on the value returned by `os.name`.

Every new component that is installed using `distutils` or a Distutils-based system will follow the same scheme to copy its file in the right places.

Python currently supports seven schemes :

- `posix_prefix` : scheme for Posix platforms like Linux or Mac OS X. This is the default scheme used when Python or a component is installed.
- `posix_home` : scheme for Posix platforms used when a `home` option is used upon installation. This scheme is used when a component is installed through Distutils with a specific home prefix.
- `posix_user` : scheme for Posix platforms used when a component is installed through Distutils and the `user` option is used. This scheme defines paths located under the user home directory.
- `nt` : scheme for NT platforms like Windows.
- `nt_user` : scheme for NT platforms, when the `user` option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths :

- `stdlib` : directory containing the standard Python library files that are not platform-specific.
- `platstdlib` : directory containing the standard Python library files that are platform-specific.
- `platlib` : directory for site-specific, platform-specific files.
- `purelib` : directory for site-specific, non-platform-specific files.
- `include` : directory for non-platform-specific header files.
- `platinclude` : directory for platform-specific header files.
- `scripts` : directory for script files.
- `data` : directory for data files.

`sysconfig` provides some functions to determine these paths.

`sysconfig.get_scheme_names()`

Return a tuple containing all schemes currently supported in `sysconfig`.

`sysconfig.get_path_names()`

Return a tuple containing all path names currently supported in `sysconfig`.

`sysconfig.get_path(name[, scheme[, vars[, expand]]])`

Return an installation path corresponding to the path `name`, from the install scheme named `scheme`.

`name` has to be a value from the list returned by `get_path_names()`.

`sysconfig` stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the `stdlib` path for the `nt` scheme is : `{base}/Lib`.

`get_path()` will use the variables returned by `get_config_vars()` to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If `scheme` is provided, it must be a value from the list returned by `get_scheme_names()`. Otherwise, the default scheme for the current platform is used.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary return by `get_config_vars()`.

If *expand* is set to `False`, the path will not be expanded using the variables.

If *name* is not found, return `None`.

`sysconfig.get_paths([scheme[, vars[, expand]]])`

Return a dictionary containing all installation paths corresponding to an installation scheme. See `get_path()` for more information.

If *scheme* is not provided, will use the default scheme for the current platform.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If *expand* is set to `false`, the paths will not be expanded.

If *scheme* is not an existing scheme, `get_paths()` will raise a `KeyError`.

29.2.3 Autres fonctions

`sysconfig.get_python_version()`

Return the MAJOR.MINOR Python version number as a string. Similar to `'%d.%d' % sys.version_info[:2]`.

`sysconfig.get_platform()`

Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by `os.uname()`), although the exact information included depends on the OS; e.g. for IRIX the architecture isn't particularly important (IRIX only runs on SGI hardware), but for Linux the kernel version isn't particularly important.

Exemples de valeurs renvoyées :

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u
- irix-5.3
- irix64-6.2

Windows will return one of :

- win-amd64 (64bit Windows on AMD64, aka x86_64, Intel64, and EM64T)
- win-ia64 (64bit Windows on Itanium)
- win32 (all others - specifically, `sys.platform` is returned)

Mac OS X can return :

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

For other non-POSIX platforms, currently just returns `sys.platform`.

`sysconfig.is_python_build()`

Return `True` if the running Python interpreter was built from source and is being run from its built location, and not from a location resulting from e.g. running `make install` or installing via a binary installer.

`sysconfig.parse_config_h(fp[, vars])`

Parse a `config.h`-style file.

fp is a file-like object pointing to the `config.h`-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

`sysconfig.get_config_h_filename()`

Return the path of `pyconfig.h`.

```
sysconfig.get_makefile_filename()
```

Return the path of Makefile.

29.2.4 Using sysconfig as a script

You can use *sysconfig* as a script with Python's *-m* option :

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
    data = "/usr/local"
    include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
    platinclude = "."
    platlib = "/usr/local/lib/python3.2/site-packages"
    platstdlib = "/usr/local/lib/python3.2"
    purelib = "/usr/local/lib/python3.2/site-packages"
    scripts = "/usr/local/bin"
    stdlib = "/usr/local/lib/python3.2"

Variables:
    AC_APPLE_UNIVERSAL_BUILD = "0"
    AIX_GENUINE_CPLUSPLUS = "0"
    AR = "ar"
    ARFLAGS = "rc"
    ...
```

This call will print in the standard output the information returned by *get_platform()*, *get_python_version()*, *get_path()* and *get_config_vars()*.

29.3 builtins — Objets natifs

Ce module fournit un accès direct aux identifiants “natifs” de Python ; par exemple, `builtins.open` est le nom complet pour la fonction native *open()*. Voir *Fonctions natives* et *Constantes natives* pour plus de documentation.

Ce module n’est normalement pas accédé explicitement par la plupart des applications, mais peut être utile dans des modules qui exposent des objets de même nom qu’une valeur native, mais pour qui le natif de même nom est aussi nécessaire. Par exemple, dans un module qui voudrait implémenter une fonction *open()* autour de la fonction native *open()*, ce module peut être utilisé directement :

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
```

(suite sur la page suivante)

(suite de la page précédente)

```
self._f = f

def read(self, count=-1):
    return self._f.read(count).upper()

# ...
```

Spécificité de l'implémentation : La plupart des modules ont `__builtins__` dans leurs globales. La valeur de `__builtins__` est classiquement soit ce module, soit la valeur de l'attribut `__dict__` du module. Puisque c'est une spécificité de CPython, ce n'est peut-être pas utilisé par toutes les autres implémentations.

29.4 `__main__` — Point d'entrée des scripts

'`__main__`' est le nom du *scope* dans lequel le code s'exécute en premier. Le nom d'un module (son `__name__`) vaut '`__main__`' lorsqu'il est lu de l'entrée standard, lorsque c'est un script, ou une invite interactive.

Un module peut découvrir s'il est exécuté dans le *scope* principal en vérifiant son `__name__`, ce qui permet typiquement d'exécuter du code lorsque le module est exécuté avec `python -m` mais pas lorsqu'il est importé :

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Pour un paquet, le même effet peut être obtenu en utilisant un module `__main__.py`, son contenu sera exécuté si le paquet est lancé via `-m`.

29.5 `warnings` — Contrôle des alertes

Code source : [Lib/warnings.py](#)

Les messages d'avertissement sont généralement émis dans les situations où il est utile d'alerter l'utilisateur d'un problème dans un programme, mais qu'il n'est pas justifié de lever une exception et de le terminer. Par exemple, on peut vouloir émettre un avertissement lorsqu'un programme utilise un module obsolète.

Les développeurs Python émettent des avertissements en appelant la fonction `warn()` définie dans ce module. (Les développeurs C utilisent `PyErr_WarnEx()` ; voir [exceptionhandling](#) pour plus d'informations).

Les messages d'avertissement sont normalement écrits sur `sys.stderr`, mais leurs effets peuvent être modifiés, il est possible d'ignorer tous les avertissements ou au contraire les transformer en exceptions. L'effet des avertissements peut varier en fonction de la catégorie d'avertissement (voir ci-dessous), de son texte et d'où il est émis. Les répétitions d'un même avertissement d'une même source sont généralement ignorés.

La gestion des avertissements se fait en deux étapes : premièrement, chaque fois qu'un avertissement est émis, le module détermine si un message doit être émis ou non ; ensuite, si un message doit être émis, il est formaté et affiché en utilisant une fonction qui peut être définie par l'utilisateur.

Un filtre (une séquence de règles) est utilisé pour décider si un message d'avertissement doit être émis ou non. Des règles peuvent être ajoutées au filtre en appelant `filterwarnings()` et remises à leur état par défaut en appelant `resetwarnings()`.

L’affichage des messages d’avertissement se fait en appelant la fonction `showwarning()`, qui peut être redéfinie ; l’implémentation par défaut formate le message en appelant `formatwarning()`, qui peut également être réutilisée par une implémentation personnalisée.

Voir aussi :

`logging.captureWarnings()` vous permet de gérer tous les avertissements avec l’infrastructure de journalisation standard.

29.5.1 Catégories d’avertissement

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings. The following warnings category classes are currently defined :

Classe	Description
<code>Warning</code>	Il s’agit de la classe de base de toutes les classes de catégories d’avertissement. C’est une sous-classe de <code>Exception</code> .
<code>UserWarning</code>	Catégorie par défaut pour <code>warn()</code> .
<code>DeprecationWarning</code>	Base category for warnings about deprecated features (ignored by default).
<code>SyntaxWarning</code>	Catégorie de base pour les avertissements concernant les syntaxes douteuses.
<code>RuntimeWarning</code>	Catégorie de base pour les avertissements concernant les fonctionnalités douteuses à l’exécution.
<code>FutureWarning</code>	Base category for warnings about constructs that will change semantically in the future.
<code>PendingDeprecationWarning</code>	Catégorie de base pour les avertissements concernant les fonctionnalités qui seront obsolètes dans le futur (ignorée par défaut).
<code>ImportWarning</code>	Catégorie de base pour les avertissements déclenchés lors de l’importation d’un module (ignoré par défaut).
<code>UnicodeWarning</code>	Catégorie de base pour les avertissements relatifs à Unicode.
<code>BytesWarning</code>	Catégorie de base pour les avertissements relatifs à <code>bytes</code> et <code>bytearray</code> .
<code>ResourceWarning</code>	Catégorie de base pour les avertissements relatifs à l’utilisation des ressources.

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.

Le code utilisateur peut définir des catégories d’avertissement supplémentaires en héritant l’une des catégories d’avertissement standard. Une catégorie d’avertissement doit toujours hériter de la classe `Warning`.

29.5.2 Le filtre des avertissements

Le filtre des avertissements contrôle si les avertissements sont ignorés, affichés ou transformés en erreurs (ce qui lève une exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications ; any specific warning is matched against each filter specification in the list in turn until a match is found ; the match determines the disposition of the match. Each entry is a tuple of the form `(action, message, category, module, lineno)`, where :

- `action` est l’une des chaînes de caractères suivantes :

Valeur	Action
"error"	transforme les avertissements correspondants en exceptions
"ignore"	ignore les avertissements correspondants
"always"	affiche toujours les avertissements correspondants
"default"	print the first occurrence of matching warnings for each location where the warning is issued
"module"	print the first occurrence of matching warnings for each module where the warning is issued
"once"	n'affiche que la première occurrence des avertissements correspondants, quel que soit l'endroit où ils se trouvent

- *message* est une chaîne de caractères contenant une expression régulière avec laquelle le début du message d'avertissement doit correspondre. L'expression est compilée pour être toujours insensible à la casse.
- *category* est une classe (une sous-classe de *Warning*) dont la catégorie d'avertissement doit être une sous-classe afin de correspondre.
- *module* est une chaîne de caractères contenant une expression régulière avec laquelle le nom du module doit correspondre. L'expression est compilée pour être sensible à la casse.
- *lineno* est le numéro de ligne d'où l'avertissement doit provenir, ou 0 pour correspondre à tous les numéros de ligne.

Puisque que la classe *Warning* hérite de la classe *Exception*, pour transformer un avertissement en erreur, il suffit de lever `category(message)`.

The warnings filter is initialized by `-W` options passed to the Python interpreter command line. The interpreter saves the arguments for all `-W` options without interpretation in `sys.warnoptions`; the *warnings* module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

Default Warning Filters

By default, Python installs several warning filters, which can be overridden by the command-line options passed to `-W` and calls to `filterwarnings()`.

- *DeprecationWarning* and *PendingDeprecationWarning*, and *ImportWarning* are ignored.
- *BytesWarning* is ignored unless the `-b` option is given once or twice; in this case this warning is either printed (`-b`) or turned into an exception (`-bb`).
- *ResourceWarning* is ignored unless Python was built in debug mode.

Modifié dans la version 3.2 : *DeprecationWarning* est maintenant ignoré par défaut en plus de *PendingDeprecationWarning*.

29.5.3 Suppression temporaire des avertissements

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning, then it is possible to suppress the warning using the *catch_warnings* context manager :

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

Dans le gestionnaire de contexte, tous les avertissements sont simplement ignorés. Ceci vous permet d'utiliser du code déclaré obsolète sans voir l'avertissement tout en ne supprimant pas l'avertissement pour un autre code qui pourrait ne pas être conscient de son utilisation de code déprécié. Remarque : ceci ne peut être garanti que dans une application utilisant

un seul fil d'exécution. Si deux ou plusieurs *threads* utilisent le gestionnaire de contexte `catch_warnings` en même temps, le comportement est indéfini.

29.5.4 Tester les avertissements

Pour tester les avertissements générés par le code, utilisez le gestionnaire de contexte `catch_warnings`. Avec lui, vous pouvez temporairement modifier le filtre d'avertissements pour faciliter votre test. Par exemple, procédez comme suit pour capturer tous les avertissements levés à vérifier :

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert isinstance(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

Vous pouvez aussi faire en sorte que tous les avertissements soient des exceptions en utilisant `error` au lieu de `always`. Il faut savoir que si un avertissement a déjà été émis à cause d'une règle `once` ou `default`, quel que soit le filtre activé, l'avertissement ne sera pas revu à moins que le registre des avertissements lié à l'avertissement ait été vidé.

A sa sortie, le gestionnaire de contexte restaure le filtre des avertissements dans l'état où il était au démarrage du contexte. Cela empêche les tests de changer le filtre d'avertissements de manière inattendue entre les tests et d'aboutir à des résultats de test indéterminés. La fonction `showwarning()` du module est également restaurée à sa valeur originale. Remarque : ceci ne peut être garanti que dans une application *mono-threadées*. Si deux ou plusieurs fils d'exécution utilisent le gestionnaire de contexte `catch_warnings` en même temps, le comportement est indéfini.

Lorsque vous testez plusieurs opérations qui provoquent le même type d'avertissement, il est important de les tester d'une manière qui confirme que chaque opération provoque un nouvel avertissement (par exemple, définissez les avertissements comme exceptions et vérifiez que les opérations provoquent des exceptions, vérifiez que la longueur de la liste des avertissements continue à augmenter après chaque opération, ou bien supprimez les entrées précédentes de la liste des avertissements avant chaque nouvelle opération).

29.5.5 Updating Code For New Versions of Python

Warnings that are only of interest to the developer are ignored by default. As such you should make sure to test your code with typically ignored warnings made visible. You can do this from the command-line by passing `-Wd` to the interpreter (this is shorthand for `-W default`). This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you simply change what argument is passed to `-W`, e.g. `-W error`. See the `-W` flag for more details on what is possible.

To programmatically do the same as `-Wd`, use :

```
warnings.simplefilter('default')
```

Make sure to execute this code as soon as possible. This prevents the registering of what warnings have been raised from unexpectedly influencing how future warnings are treated.

Having certain warnings ignored by default is done to prevent a user from seeing warnings that are only of interest to the developer. As you do not necessarily have control over what interpreter a user uses to run their code, it is possible that a new version of Python will be released between your release cycles. The new interpreter release could trigger new warnings in your code that were not there in an older interpreter, e.g. `DeprecationWarning` for a module that you are using. While you as a developer want to be notified that your code is using a deprecated module, to a user this information is essentially noise and provides no benefit to them.

The `unittest` module has been also updated to use the `'default'` filter while running tests.

29.5.6 Fonctions disponibles

`warnings.warn(message, category=None, stacklevel=1, source=None)`

Émet, ignore, ou transforme en exception un avertissement. L'argument `category`, s'il est donné, doit être une classe de catégorie d'avertissement (voir ci-dessus); et vaut par défaut `UserWarning`. Aussi `message` peut être une instance de `Warning`, auquel cas `*category*` sera ignoré et `message.__class__` sera utilisé. Dans ce cas, le texte du message sera str(message). Cette fonction lève une exception si cet avertissement particulier émis est transformé en erreur par le filtre des avertissements, voir ci-dessus. L'argument stacklevel peut être utilisé par les fonctions wrapper écrites en Python, comme ceci :`

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

Fait en sorte que l'avertissement se réfère à l'appelant de `deprecation()` plutôt qu'à la source de `deprecation()` elle-même (puisque celle-ci irait à l'encontre du but du message d'avertissement).

`source`, s'il est fourni, est l'objet détruit qui a émis un `ResourceWarning`.

Modifié dans la version 3.6 : Ajout du paramètre `source`.

`warnings.warn_explicit(message, category, filename, lineno, module=None, registry=None, module_globals=None, source=None)`

Il s'agit d'une interface de bas niveau pour la fonctionnalité de `warn()`, en passant explicitement le message, la catégorie, le nom de fichier et le numéro de ligne, et éventuellement le nom du module et le registre (qui devrait être le dictionnaire `__warningregistry__` du module). Le nom de module par défaut est le nom de fichier sans `.py`; si aucun registre n'est passé, l'avertissement n'est jamais supprimé. `message` doit être une chaîne de caractères et `category` une sous-classe de `Warning` ou `message` peut être une instance de `Warning`, auquel cas `category` sera ignoré.

`module_globals`, s'il est fourni, doit être l'espace de nommage global utilisé par le code pour lequel l'avertissement est émis. (Cet argument est utilisé pour afficher les sources des modules trouvés dans les fichiers zip ou d'autres sources d'importation hors du système de fichiers).

`source`, s'il est fourni, est l'objet détruit qui a émis un `ResourceWarning`.

Modifié dans la version 3.6 : Ajout du paramètre `source`.

`warnings.showwarning(message, category, filename, lineno, file=None, line=None)`

Écrit un avertissement dans un fichier. L'implémentation par défaut appelle `format_warning(message, category, filename, lineno, line)` et écrit la chaîne résultante dans `*file*`, qui par défaut est `sys.stderr`. Vous pouvez remplacer cette fonction par n'importe quel appellable en l'affectant à `warnings.showwarning`. `line` est une ligne de code source à inclure dans le message d'avertissement; si `line` n'est pas fourni, `show warning()` essaiera de lire la ligne spécifiée par `filename` et `lineno`.

`warnings.formatwarning(message, category, filename, lineno, line=None)`

Formate un avertissement de la manière standard. Ceci renvoie une chaîne pouvant contenir des retours à la ligne se termine par un retour à la ligne. `line` est une ligne de code source à inclure dans le message d'avertissement; si `line` n'est pas fourni, `formatwarning()` essaiera de lire la ligne spécifiée par `filename` et `lineno`.

`warnings.filterwarnings(action, message="", category=Warning, module="", lineno=0, append=False)`

Insère une entrée dans la liste de `warning filter specifications`. L'entrée est insérée à l'avant par défaut; si `append` est

vrai, elle est insérée à la fin. Il vérifie le type des arguments, compile les expressions régulières *message* et *module*, et les insère sous forme de tuple dans la liste des filtres d'avertissements. Les entrées plus proches du début de la liste ont priorité sur les entrées plus loin dans la liste. Les arguments omis ont par défaut une valeur qui correspond à tout.

`warnings.simplefilter(action, category=Warning, lineno=0, append=False)`

Insère une entrée simple dans la liste de *spécifications du filtre d'avertissements*. La signification des paramètres de fonction est la même que pour `filterwarnings()`, mais les expressions régulières ne sont pas nécessaires car le filtre inséré correspond toujours à n'importe quel message dans n'importe quel module tant que la catégorie et le numéro de ligne correspondent.

`warnings.resetwarnings()`

Réinitialise le filtre des avertissements. Ceci supprime l'effet de tous les appels précédents à `filterwarnings()`, y compris celui de l'option `-W` des options de ligne de commande et des appels à `simplefilter()`.

29.5.7 Gestionnaires de contexte disponibles

`class warnings.catch_warnings(*, record=False, module=None)`

Un gestionnaire de contexte qui copie et, à la sortie, restaure le filtre des avertissements et la fonction `showwarning()`. Si l'argument *record* est *False* (par défaut), le gestionnaire de contexte retourne *None* en entrant. Si *record* est *True*, une liste est renvoyée qui est progressivement remplie d'objets comme vus par une fonction custom `showwarning'` (qui supprime également la sortie vers `sys.stdout`()`). Chaque objet de la liste a des attributs avec les mêmes noms que les arguments de `showwarning()`.

L'argument *module* prend un module qui sera utilisé à la place du module renvoyé lors de l'importation `warnings` dont le filtre sera protégé. Cet argument existe principalement pour tester le module `warnings` lui-même.

Note : Le gestionnaire `catch_warnings` fonctionne en remplaçant puis en restaurant plus tard la fonction `showwarning()` du module et la liste interne des spécifications du filtre. Cela signifie que le gestionnaire de contexte modifie l'état global et n'est donc pas prévisible avec plusieurs fils d'exécution.

29.6 contextlib — Utilities for with-statement contexts

Code source : [Lib/contextlib.py](#)

Ce module fournit des utilitaires pour les tâches impliquant le mot-clé `with`. Pour plus d'informations voir aussi *Le type gestionnaire de contexte* et `context-managers`.

29.6.1 Utilitaires

Fonctions et classes fournies :

`class contextlib.AbstractContextManager`

Classe mère abstraite pour les classes qui implémentent les méthodes `object.__enter__()` et `object.__exit__()`. Une implémentation par défaut de `object.__enter__()` est fournie, qui renvoie `self`, et `object.__exit__()` est une méthode abstraite qui renvoie `None` par défaut. Voir aussi la définition de *Le type gestionnaire de contexte*.

Nouveau dans la version 3.6.

`@contextlib.contextmanager`

Cette fonction est un *decorator* qui peut être utilisé pour définir une fonction fabriquant des gestionnaires de contexte à utiliser avec `with`, sans nécessiter de créer une classe ou des méthodes `__enter__()` et `__exit__()` séparées.

While many objects natively support use in `with` statements, sometimes a resource needs to be managed that isn't a context manager in its own right, and doesn't implement a `close()` method for use with `contextlib.closing`

An abstract example would be the following to ensure correct resource management :

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwargs):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwargs)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)

>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

The function being decorated must return a *generator*-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the `with` statement's `as` clause, if any.

At the point where the generator yields, the block nested in the `with` statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a `try...except...finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the `with` statement that the exception has been handled, and execution will resume with the statement immediately following the `with` statement.

Le décorateur `contextmanager()` utilise la classe `ContextDecorator` afin que les gestionnaires de contexte qu'il crée puissent être utilisés aussi bien en tant que décorateurs qu'avec des instructions `with`. Quand vous l'utilisez comme décorateur, une nouvelle instance du générateur est créée à chaque appel de la fonction (cela permet aux gestionnaires de contexte à usage unique créés par `contextmanager()` de remplir la condition de pouvoir être invoqués plusieurs fois afin d'être utilisés comme décorateurs).

Modifié dans la version 3.2 : Utilisation de la classe `ContextDecorator`.

`contextlib.closing(thing)`

Renvoie un gestionnaire de contexte qui ferme *thing* à la fin du bloc. C'est essentiellement équivalent à :

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

Et cela vous permet d'écrire du code tel que :

```
from contextlib import closing
from urllib.request import urlopen
```

(suite sur la page suivante)

(suite de la page précédente)

```
with closing(urlopen('http://www.python.org')) as page:
    for line in page:
        print(line)
```

sans besoin de fermer explicitement `page`. Même si une erreur survient, `page.close()` est appelée à la fermeture du bloc `with`.

`contextlib.suppress(*exceptions)`

Renvoie un gestionnaire de contexte qui supprime toutes les exceptions spécifiées si elles surviennent dans le corps du bloc `with`, et reprend l'exécution sur la première instruction qui suit la fin du bloc `with`.

Comme pour tous les mécanismes qui suppriment complètement les exceptions, ce gestionnaire de contexte doit seulement être utilisé pour couvrir des cas très spécifiques d'erreurs où il est certain que continuer silencieusement l'exécution du programme est la bonne chose à faire.

Par exemple :

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

Ce code est équivalent à :

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

Ce gestionnaire de contexte est *réentrant*.

Nouveau dans la version 3.4.

`contextlib.redirect_stdout(new_target)`

Gestionnaire de contexte servant à rediriger temporairement `sys.stdout` vers un autre fichier ou objet fichier-compatible.

Cet outil ajoute une certaine flexibilité aux fonctions ou classes existantes dont la sortie est envoyée vers la sortie standard.

Par exemple, la sortie de `help()` est normalement envoyée vers `sys.stdout`. Vous pouvez capturer cette sortie dans une chaîne de caractères en la redirigeant vers un objet `io.StringIO` :

```
f = io.StringIO()
with redirect_stdout(f):
    help(pow)
s = f.getvalue()
```

Pour envoyer la sortie de `help()` vers un fichier sur le disque, redirigez-la sur un fichier normal :

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

Pour envoyer la sortie de `help()` sur `sys.stderr` :

```
with redirect_stdout(sys.stderr):
    help(pow)
```

Notez que l'effet de bord global sur `sys.stdout` signifie que ce gestionnaire de contexte n'est pas adapté à une utilisation dans le code d'une bibliothèque ni dans la plupart des applications à plusieurs fils d'exécution. Aussi, cela n'a pas d'effet sur la sortie des sous-processus. Cependant, cela reste une approche utile pour beaucoup de scripts utilitaires.

Ce gestionnaire de contexte est *réentrant*.

Nouveau dans la version 3.4.

`contextlib.redirect_stderr(new_target)`

Similaire à `redirect_stdout()` mais redirige `sys.stderr` vers un autre fichier ou objet fichier-compatible.

Ce gestionnaire de contexte est *réentrant*.

Nouveau dans la version 3.5.

class `contextlib.ContextDecorator`

Une classe mère qui permet à un gestionnaire de contexte d'être aussi utilisé comme décorateur.

Les gestionnaires de contexte héritant de `ContextDecorator` doivent implémenter `__enter__` et `__exit__` comme habituellement. `__exit__` conserve sa gestion optionnelle des exceptions même lors de l'utilisation en décorateur.

`ContextDecorator` est utilisé par `contextmanager()`, donc vous bénéficiez automatiquement de cette fonctionnalité.

Exemple de `ContextDecorator` :

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False

>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

Ce changement est simplement un sucre syntaxique pour les constructions de la forme suivante :

```
def f():
    with cm():
        # Do stuff
```

`ContextDecorator` vous permet d'écrire à la place :

```
@cm()
def f():
    # Do stuff
```

Cela éclaircit le fait que `cm` s'applique à la fonction entière, et pas seulement à un morceau en particulier (et gagner un niveau d'indentation est toujours appréciable).

Les gestionnaires de contexte existants qui ont déjà une classe mère peuvent être étendus en utilisant `ContextDecorator` comme une *mixin* :

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

Note : As the decorated function must be able to be called multiple times, the underlying context manager must support use in multiple `with` statements. If this is not the case, then the original construct with the explicit `with` statement inside the function should be used.

Nouveau dans la version 3.2.

class `contextlib.ExitStack`

Gestionnaire de contexte conçu pour simplifier le fait de combiner programmatiquement d'autres gestionnaires de contexte et fonctions de nettoyage, spécifiquement ceux qui sont optionnels ou pilotés par des données d'entrée.

Par exemple, un ensemble de fichiers peut facilement être géré dans une unique instruction *with* comme suit :

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

Chaque instance maintient une pile de fonctions de rappels (*callbacks*) enregistrées qui sont appelées en ordre inverse quand l'instance est fermée (explicitement ou implicitement à la fin d'un bloc `with`). Notez que ces fonctions ne sont *pas* invoquées implicitement quand l'instance de la pile de contextes est collectée par le ramasse-miettes.

Ce modèle de pile est utilisé afin que les gestionnaires de contexte qui acquièrent leurs ressources dans leur méthode `__init__` (tels que les objets-fichiers) puissent être gérés correctement.

Comme les fonctions de rappel enregistrées sont invoquées dans l'ordre inverse d'enregistrement, cela revient au même que si de multiples blocs `with` imbriqués avaient été utilisés avec l'ensemble de fonctions enregistrées. Cela s'étend aussi à la gestion d'exceptions — si une fonction de rappel intérieure supprime ou remplace une exception, alors les fonctions extérieures reçoivent des arguments basés sur ce nouvel état.

C'est une *API* relativement bas-niveau qui s'occupe de dérouler correctement la pile des appels de sortie. Elle fournit une base adaptée pour des gestionnaires de contexte de plus haut niveau qui manipulent la pile de sortie de manière spécifique à l'application.

Nouveau dans la version 3.3.

enter_context (*cm*)

Entre dans un nouveau gestionnaire de contexte et ajoute sa méthode `__exit__()` à la pile d'appels. La valeur de retour est le résultat de la méthode `__enter__()` du gestionnaire de contexte donné.

Ces gestionnaires de contexte peuvent supprimer des exceptions comme ils le feraient normalement s'ils étaient utilisés directement derrière une instruction `with`.

push (*exit*)

Ajoute la méthode `__exit__()` d'un gestionnaire de contexte à la pile d'appels.

Comme `__enter__` n'est *pas* invoquée, cette méthode peut être utilisée pour couvrir une partie de l'implémentation de `__enter__()` avec la propre méthode `__exit__()` d'un gestionnaire de contexte.

Si l'argument passé n'est pas un gestionnaire de contexte, la méthode assume qu'il s'agit d'une fonction de rappel avec la même signature que la méthode `__exit__()` des gestionnaires de contexte pour l'ajouter directement à la pile d'appels.

En retournant des valeurs vraies, ces fonctions peuvent supprimer des exceptions de la même manière que le peuvent les méthodes `__exit__()` des gestionnaires de contexte.

L'objet passé en paramètre est renvoyé par la fonction, ce qui permet à la méthode d'être utilisée comme décorateur de fonction.

callback (*callback*, **args*, ***kwargs*)

Accepte une fonction arbitraire et ses arguments et les ajoute à la pile des fonctions de rappel.

À la différence des autres méthodes, les fonctions de rappel ajoutées de cette manière ne peuvent pas supprimer les exceptions (puisque'elles ne reçoivent jamais les détails de l'exception).

La fonction passée en paramètre est renvoyée par la méthode, ce qui permet à la méthode d'être utilisée comme décorateur de fonction.

pop_all ()

Transfère la pile d'appels à une nouvelle instance de `ExitStack` et la renvoie. Aucune fonction de rappel n'est invoquée par cette opération — à la place, elles sont dorénavant invoquées quand la nouvelle pile sera close (soit explicitement soit implicitement à la fin d'un bloc `with`).

Par exemple, un groupe de fichiers peut être ouvert comme une opération « tout ou rien » comme suit :

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close ()

Déroule immédiatement la pile d'appels, invoquant les fonctions de rappel dans l'ordre inverse d'enregistrement. Pour chaque gestionnaire de contexte et fonction de sortie enregistré, les arguments passés indiqueront qu'aucune exception n'est survenue.

29.6.2 Exemples et Recettes

Cette section décrit quelques exemples et recettes pour décrire une utilisation réelle des outils fournis par `contextlib`.

Gérer un nombre variable de gestionnaires de contexte

Le cas d'utilisation primaire de `ExitStack` est celui décrit dans la documentation de la classe : gérer un nombre variable de gestionnaires de contexte et d'autres opérations de nettoyage en une unique instruction `with`. La variabilité peut venir du nombre de gestionnaires de contexte voulus découlant d'une entrée de l'utilisateur (comme ouvrir une collection spécifique de fichiers de l'utilisateur), ou de certains gestionnaires de contexte qui peuvent être optionnels :

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
```

(suite sur la page suivante)

(suite de la page précédente)

```

special = acquire_special_resource()
stack.callback(release_special_resource, special)
# Perform operations that use the acquired resources

```

Comme montré, *ExitStack* rend aussi assez facile d'utiliser les instructions `with` pour gérer des ressources arbitraires qui ne gèrent pas nativement le protocole des gestionnaires de contexte.

Simplifying support for single optional context managers

In the specific case of a single optional context manager, *ExitStack* instances can be used as a « do nothing » context manager, allowing a context manager to easily be omitted without affecting the overall structure of the source code :

```

def debug_trace(details):
    if __debug__:
        return TraceContext(details)
    # Don't do anything special with the context in release mode
    return ExitStack()

with debug_trace():
    # Suite is traced in debug mode, but runs normally otherwise

```

Attraper des exceptions depuis les méthodes `__enter__`

Il est occasionnellement souhaitable d'attraper les exceptions depuis l'implémentation d'une méthode `__enter__`, sans attraper par inadvertance les exceptions du corps de l'instruction `with` ou de la méthode `__exit__` des gestionnaires de contexte. En utilisant *ExitStack*, les étapes du protocole des gestionnaires de contexte peuvent être légèrement séparées pour permettre le code suivant :

```

stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case

```

Avoir à faire cela est en fait surtout utile pour indiquer que l'API sous-jacente devrait fournir une interface directe de gestion des ressources à utiliser avec les instructions `try/except/finally`, mais que toutes les API ne sont pas bien conçues dans cet objectif. Quand un gestionnaire de contexte est la seule API de gestion des ressources fournie, alors *ExitStack* peut rendre plus facile la gestion de plusieurs situations qui ne peuvent pas être traitées directement dans une instruction `with`.

Nettoyer dans une méthode `__enter__`

Comme indiqué dans la documentation de `ExitStack.push()`, cette méthode peut être utile pour nettoyer une ressource déjà allouée si les dernières étapes de l'implémentation de `__enter__()` échouent.

Voici un exemple de gestionnaire de contexte qui reçoit des fonctions d'acquisition de ressources et de libération, avec une méthode de validation optionnelle, et qui les adapte au protocole des gestionnaires de contexte :

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

Remplacer un `try-finally` avec une option variable

Un modèle que vous rencontrerez parfois est un bloc `try-finally` avec une option pour indiquer si le corps de la clause `finally` doit être exécuté ou non. Dans sa forme la plus simple (qui ne peut pas déjà être gérée avec juste une clause `except`), cela ressemble à :

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```


Comme avec `n'importe quel code basé sur une instruction try`, cela peut poser problème pour le développement et la revue, parce que beaucoup de codes d'installation et de nettoyage peuvent finir par être séparés par des sections de code arbitrairement longues.

`ExitStack` rend possible de plutôt enregistrer une fonction de rappel pour être exécutée à la fin d'une instruction `with`, et décider ensuite de passer l'exécution de cet appel :

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

Cela permet de rendre explicite dès le départ le comportement de nettoyage attendu, plutôt que de nécessiter une option séparée.

Si une application particulière utilise beaucoup ce modèle, cela peut-être simplifié encore plus au moyen d'une petite classe d'aide :

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, *args, **kwargs):
        super(Callback, self).__init__()
        self.callback(callback, *args, **kwargs)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

Si le nettoyage de la ressource n'est pas déjà soigneusement embarqué dans une fonction autonome, il est possible d'utiliser le décorateur `ExitStack.callback()` pour déclarer la fonction de nettoyage de ressource en avance :

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

Dû au fonctionnement du protocole des décorateurs, une fonction déclarée ainsi ne peut prendre aucun paramètre. À la place, les ressources à libérer doivent être récupérées depuis l'extérieur comme des variables de fermeture (*closure*).

Utiliser un gestionnaire de contexte en tant que décorateur de fonction

ContextDecorator rend possible l'utilisation d'un gestionnaire de contexte à la fois ordinairement avec une instruction `with` ou comme un décorateur de fonction.

Par exemple, il est parfois utile d'emballer les fonctions ou blocs d'instructions avec un journaliseur qui pourrait suivre le temps d'exécution entre l'entrée et la sortie. Plutôt qu'écrire à la fois un décorateur et un gestionnaire de contexte pour la même tâche, hériter de *ContextDecorator* fournit les deux fonctionnalités en une seule définition :

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Les instances de cette classe peuvent être utilisées comme gestionnaires de contexte :

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

Et comme décorateurs de fonctions :

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Notez qu'il y a une autre limitation en utilisant les gestionnaires de contexte comme décorateurs : il n'y a aucune manière d'accéder à la valeur de retour de `__enter__()`. Si cette valeur est nécessaire, il faut utiliser explicitement une instruction `with`.

Voir aussi :

PEP 343 - The « with » statement La spécification, les motivations et des exemples de l'instruction `with` en Python.

29.6.3 Gestionnaires de contexte à usage unique, réutilisables et réentrants

La plupart des gestionnaires de contexte sont écrits d'une manière qui ne leur permet que d'être utilisés une fois avec une instruction `with`. Ces gestionnaires de contexte à usage unique doivent être recréés chaque fois qu'ils sont utilisés — tenter de les utiliser une seconde fois lève une exception ou ne fonctionne pas correctement.

Cette limitation commune signifie qu'il est généralement conseillé de créer les gestionnaires de contexte directement dans l'en-tête du bloc `with` où ils sont utilisés (comme montré dans tous les exemples d'utilisation au-dessus).

Les fichiers sont un exemple de gestionnaires de contexte étant effectivement à usage unique, puisque la première instruction `with` ferme le fichier, empêchant d'autres opérations d'entrée/sortie d'être exécutées sur ce fichier.

Les gestionnaires de contexte créés avec `contextmanager()` sont aussi à usage unique, et se plaindront du fait que le générateur sous-jacent ne produise plus de valeur si vous essayez de les utiliser une seconde fois :

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
...
RuntimeError: generator didn't yield
```

Gestionnaires de contexte réentrants

More sophisticated context managers may be « reentrant ». These context managers can not only be used in multiple with statements, but may also be used *inside* a with statement that is already using the same context manager.

`threading.RLock` est un exemple de gestionnaire de contexte réentrant, comme le sont aussi `suppress()` et `redirect_stdout()`. Voici un très simple exemple d'utilisation réentrante :

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

Les exemples plus réels de réentrance sont susceptibles d'invoquer plusieurs fonctions s'entre-appelant, et donc être bien plus compliqués que cet exemple.

Notez aussi qu'être réentrant ne signifie *pas* être *thread safe*. `redirect_stdout()`, par exemple, n'est définitivement pas *thread safe*, puisqu'il effectue des changements globaux sur l'état du système en branchant `sys.stdout` sur différents flux.

Gestionnaires de contexte réutilisables

D'autres gestionnaires de contexte que ceux à usage unique et les réentrants sont les gestionnaires de contexte « réutilisables » (ou, pour être plus explicite, « réutilisables mais pas réentrants », puisque les gestionnaires de contexte réentrants sont aussi réutilisables). Ces gestionnaires de contexte sont conçus afin d'être utilisés plusieurs fois, mais échoueront (ou ne fonctionnent pas correctement) si l'instance de gestionnaire de contexte référencée a déjà été utilisée dans une instruction *with* englobante.

`threading.Lock` est un exemple de gestionnaire de contexte réutilisable mais pas réentrant (pour un verrou réentrant, il faut à la place utiliser `threading.RLock`).

Un autre exemple de gestionnaire de contexte réutilisable mais pas réentrant est `ExitStack`, puisqu'il invoque *toutes* les fonctions de rappel actuellement enregistrées en quittant l'instruction *with*, sans regarder où ces fonctions ont été ajoutées :

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

Comme le montre la sortie de l'exemple, réutiliser une simple pile entre plusieurs instructions *with* fonctionne correctement, mais essayer de les imbriquer fait que la pile est vidée à la fin du *with* le plus imbriqué, ce qui n'est probablement pas le comportement voulu.

Pour éviter ce problème, utilisez des instances différentes de `ExitStack` plutôt qu'une seule instance :

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

29.7 abc — Classes de Base Abstraites

Code source : [Lib/abc.py](#)

Le module fournit l'infrastructure pour définir les *classes de bases abstraites* (*Abstract Base Class* ou *ABC* en anglais) en Python, tel qu'indiqué dans la [PEP 3119](#) ; voir la PEP pour la raison de son ajout à Python. (Voir également la [PEP 3141](#) et le module `numbers` pour ce qui concerne la hiérarchie de types pour les nombres basés sur les classes de base abstraites). Par la suite nous utiliserons l'abréviation *ABC* (*Abstract Base Class*) pour désigner une classe de base abstraite.

The `collections` module has some concrete classes that derive from ABCs ; these can, of course, be further derived. In addition, the `collections.abc` submodule has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, if it is hashable or if it is a mapping.

Ce module fournit la métaclasse `ABCMeta` pour définir les ABC ainsi que la classe d'aide `ABC`, cette dernière permettant de définir des ABC en utilisant l'héritage :

class `abc.ABC`

Classe d'aide qui a `ABCMeta` pour métaclasse. Avec cette classe, une ABC peut être créée simplement en héritant de `ABC`, ce qui permet d'éviter l'utilisation parfois déroutante de métaclasse, par exemple :

```
from abc import ABC

class MyABC(ABC):
    pass
```

Il est à noter que le type de `ABC` reste `ABCMeta`. En conséquence, hériter de `ABC` nécessite les précautions habituelles concernant l'utilisation de métaclasses : l'utilisation d'héritage multiple peut entraîner des conflits de métaclasses. Il est également possible de définir une ABC en passant l'argument nommé `metaclass` et en utilisant `ABCMeta` directement, par exemple :

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

Nouveau dans la version 3.4.

class `abc.ABCMeta`

Métaclasse pour définir des classes de base abstraites (ABC).

Utilisez cette métaclasse pour créer une ABC. Il est possible d'hériter d'une ABC directement, cette classe de base abstraite fonctionne alors comme une classe *mixin*. Vous pouvez également enregistrer une classe concrète sans lien (même une classe native) et des ABC comme « sous-classes virtuelles » – celles-ci et leur descendantes seront considérées comme des sous-classes de la classe de base abstraite par la fonction native `issubclass()`, mais les ABC enregistrées n'apparaîtront pas dans leur ordre de résolution des méthodes (*MRO* pour *Method Resolution Order* en anglais). Les implémentations de méthodes définies par l'ABC ne seront pas appelable (pas même via `super()`).¹

Les classes dont la métaclasse est `ABCMeta` possèdent les méthodes suivantes :

register (*subclass*)

Enregistrer *subclass* en tant que sous-classe virtuelle de cette ABC. Par exemple :

```
from abc import ABC

class MyABC(ABC):
    pass
```

(suite sur la page suivante)

1. Les développeurs C++ noteront que le concept Python de classe de base virtuelle (*virtual base class*) n'est pas le même que celui de C++.

(suite de la page précédente)

```
MyABC.register(tuple)

assert isinstance(tuple, MyABC)
assert isinstance((), MyABC)
```

Modifié dans la version 3.3 : Renvoie la sous-classe enregistrée pour permettre l'utilisation en tant que décorateur de classe.

Modifié dans la version 3.4 : Pour détecter les appels à `register()`, vous pouvez utiliser la fonction `get_cache_token()`.

Vous pouvez également redéfinir cette méthode dans une ABC :

`__subclasshook__` (*subclass*)

(Doit être définie en tant que méthode de classe.)

Vérifie si *subclass* est considérée comme une sous-classe de cette ABC. Cela signifie que vous pouvez personnaliser le comportement de `issubclass` sans nécessiter d'appeler `register()` pour chacune des classes que vous souhaitez considérer comme sous-classe de l'ABC. (Cette méthode de classe est appelée par la méthode `__subclasscheck__()` de la classe de base abstraite).

Cette méthode doit renvoyer `True`, `False` ou `NotImplemented`. Si elle renvoie `True`, *subclass* est considérée comme sous-classe de cette ABC. Si elle renvoie `False`, la *subclass* n'est pas considérée une sous-classe de cette ABC même si elle l'aurait été en temps normal. Si elle renvoie `NotImplemented`, la vérification d'appartenance à la sous-classe continue via le mécanisme habituel.

Pour une illustration de ces concepts, voir cet exemple de définition de ABC :

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
            return NotImplemented

MyIterable.register(Foo)
```

L'ABC `MyIterable` définit la méthode d'itération `__iter__()` comme méthode abstraite. L'implémentation qui lui est donnée ici peut être appelée depuis les sous-classes. La méthode `get_iterator()` fait également partie de la classe de base abstraite `MyIterable`, mais elle n'a pas à être redéfinie dans les classes dérivées non-abstraites.

La méthode de classe `__subclasshook__()` définie ici dit que toute classe qui possède la méthode `__iter__()` dans son `__dict__` (ou dans une de ses classes de base, accédée via la liste `__mro__`) est

considérée également comme un `MyIterable`.

Enfin, la dernière ligne fait de `Foo` une sous-classe virtuelle de `MyIterable`, même si cette classe ne définit pas de méthode `__iter__()` (elle utilise l'ancien protocole d'itération qui se définit en termes de `__len__()` et `__getitem__()`). A noter que cela ne rendra pas le `get_iterator` de `MyIterable` disponible comme une méthode de `Foo`, `get_iterator` est donc implémenté séparément.

Le module `abc` fournit aussi le décorateur :

`@abc.abstractmethod`

Un décorateur marquant les méthodes comme abstraites.

Utiliser ce décorateur nécessite que la métaclasse de la classe soit `ABCMeta` ou soit dérivée de celle-ci. Une classe qui possède une méta-classe dérivée de `ABCMeta` ne peut pas être instanciée à moins que toutes ses méthodes et propriétés abstraites soient redéfinies. Les méthodes abstraites peuvent être appelées en utilisant n'importe quel des mécanismes d'appel à "super". `abstractmethod()` peut être utilisée pour déclarer des méthodes abstraites pour les propriétés et descripteurs.

Python ne gère pas l'ajout dynamique de méthodes abstraites à une classe, il n'est pas non plus possible de modifier l'état d'abstraction d'une méthode ou d'une classe une fois celle-ci créée. `abstractmethod()` n'affecte que les sous-classes dérivées utilisant l'héritage classique. Les « sous-classes virtuelles » enregistrées avec la méthode `register()` de l'ABC ne sont pas affectées.

Quand le décorateur `abstractmethod()` est utilisé en même temps que d'autres descripteurs de méthodes, il doit être appliqué en tant que décorateur le plus interne. Voir les exemples d'utilisation suivants :

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, ...):
        ...

    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...

    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

In order to correctly interoperate with the abstract base class machinery, the descriptor must identify itself as abstract using `__isabstractmethod__`. In general, this attribute should be `True` if any of the methods used to compose the descriptor are abstract. For example, Python's built-in `property` does the equivalent of :

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                    f in (self._fget, self._fset, self._fdel))
```

Note : Contrairement aux méthodes abstraites Java, ces méthodes abstraites peuvent être implémentées. Cette implémentation peut être appelée via le mécanisme `super()` depuis la classe qui la redéfinit. C'est typiquement utile pour y appeler `super` et ainsi coopérer correctement dans un environnement utilisant de l'héritage multiple.

Le module `abc` gère également les décorateurs historiques suivants :

`@abc.abstractclassmethod`

Nouveau dans la version 3.2.

Obsolète depuis la version 3.3 : Il est désormais possible d'utiliser `classmethod` avec `abstractmethod()`, cela rend ce décorateur redondant.

Sous-classe du décorateur natif `classmethod()` qui indique une méthode de classe (`classmethod`) abstraite. En dehors de cela, est similaire à `abstractmethod()`.

Ce cas spécial est obsolète car le décorateur `classmethod()` est désormais correctement identifié comme abstrait quand il est appliqué à une méthode abstraite :

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
```

`@abc.abstractstaticmethod`

Nouveau dans la version 3.2.

Obsolète depuis la version 3.3 : Il est désormais possible d'utiliser `staticmethod` avec `abstractmethod()`, cela rend ce décorateur redondant.

Sous-classe du décorateur natif `classmethod()` qui indique une méthode statique (`staticmethod`) abstraite. En dehors de cela, est similaire à `abstractmethod()`.

Ce cas spécial est obsolète car le décorateur `staticmethod()` est désormais correctement identifié comme abstrait quand appliqué à une méthode abstraite :

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...
```

`@abc.abstractproperty`

Obsolète depuis la version 3.3 : Il est désormais possible d'utiliser `property`, `property.getter()`, `property.setter()` et `property.deleter()` avec `abstractmethod()`, ce qui rend ce décorateur redondant.

Sous-classe de `property()`, qui indique une propriété abstraite.

Ce cas spécial est obsolète car le décorateur `property()` est désormais correctement identifié comme abstrait quand appliqué à une méthode abstraite :

```
class C(ABC):
    @property
```

(suite sur la page suivante)

(suite de la page précédente)

```
@abstractmethod
def my_abstract_property(self):
    ...
```

L'exemple ci-dessus définit une propriété en lecture seule. Vous pouvez également définir une propriété en lecture-écriture abstraite en indiquant une ou plusieurs des méthodes sous-jacentes comme abstraite :

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

Si seuls certains composants sont abstraits, seuls ces composants abstraits nécessitent d'être mis à jour pour créer une propriété concrète dans une sous-classe :

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

Le module `abc` fournit également la fonction suivante :

`abc.get_cache_token()`

Renvoie le jeton de cache (*cache token*) de l'ABC.

Le jeton est un objet opaque (qui implémente le test d'égalité) qui identifie la version actuelle du cache de l'ABC pour les sous-classes virtuelles. Le jeton change avec chaque appel à `ABCMeta.register()` sur n'importe quelle ABC.

Nouveau dans la version 3.4.

Notes

29.8 atexit — Gestionnaire de fin de programme

Le module `atexit` définit des fonctions pour inscrire et désinscrire des fonctions de nettoyage. Les fonctions ainsi inscrites sont automatiquement exécutées au moment de l'arrêt normal de l'interpréteur. `atexit` exécute ces fonctions dans l'ordre inverse dans lequel elles ont été inscrites ; si vous inscrivez A, B, et C, au moment de l'arrêt de l'interpréteur elles seront exécutées dans l'ordre C, B, A.

Note : Les fonctions inscrites via ce module ne sont pas appelées quand le programme est tué par un signal non géré par Python, quand une erreur fatale interne de Python est détectée, ou quand `os._exit()` est appelé.

`atexit.register(func, *args, **kwargs)`

Inscrit `func` comme une fonction à exécuter au moment de l'arrêt de l'interpréteur. Tout argument optionnel qui doit être passé à `func` doit être passé comme argument à `register()`. Il est possible d'inscrire les mêmes fonctions et arguments plus d'une fois.

Lors d'un arrêt normal du programme (par exemple, si `sys.exit()` est appelée ou l'exécution du module principal se termine), toutes les fonctions inscrites sont appelées, dans l'ordre de la dernière arrivée, première servie. La

supposition est que les modules les plus bas niveau vont normalement être importés avant les modules haut niveau et ainsi être nettoyés en dernier.

Si une exception est levée durant l'exécution du gestionnaire de fin de programme, une trace d'appels est affichée (à moins que `SystemExit` ait été levée) et les informations de l'exception sont sauvegardées. Une fois que tous les gestionnaires de fin de programme ont eu une chance de s'exécuter, la dernière exception à avoir été levée l'est de nouveau.

Cette fonction renvoie *func*, ce qui rend possible de l'utiliser en tant que décorateur.

`atexit.unregister(func)`

Retire *func* de la liste des fonctions à exécuter à l'arrêt de l'interpréteur. Après avoir appelé `unregister()`, *func* est garantie de ne pas être appelée à l'arrêt de l'interpréteur, même si elle a été inscrite plus d'une fois. `unregister()` ne fait rien et reste muette dans le cas où *func* n'a pas été inscrite précédemment.

Voir aussi :

Module `readline` Un exemple utile de l'usage de `atexit` pour lire et écrire des fichiers d'historique `readline`.

29.8.1 Exemple avec `atexit`

Le simple exemple suivant démontre comment un module peut initialiser un compteur depuis un fichier quand il est importé, et sauve la valeur mise à jour du compteur automatiquement quand le programme se termine, sans avoir besoin que l'application fasse un appel explicite dans ce module au moment de l'arrêt de l'interpréteur.

```
try:
    with open("counterfile") as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open("counterfile", "w") as outfile:
        outfile.write("%d" % _count)

import atexit
atexit.register(savecounter)
```

Les arguments positionnels et par mot-clé peuvent aussi être passés à `register()` afin d'être repassés à la fonction inscrite lors de son appel :

```
def goodbye(name, adjective):
    print('Goodbye, %s, it was %s to meet you.' % (name, adjective))

import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

Utilisation en tant que *décorateur* :

```
import atexit
```

(suite sur la page suivante)

(suite de la page précédente)

```
@atexit.register
def goodbye():
    print("You are now leaving the Python sector.")
```

Ceci fonctionne uniquement avec des fonctions qui peuvent être appelées sans argument.

29.9 `traceback` — Print or retrieve a stack traceback

Source code : [Lib/traceback.py](#)

This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, such as in a « wrapper » around the interpreter.

The module uses traceback objects — this is the object type that is stored in the `sys.last_traceback` variable and returned as the third item from `sys.exc_info()`.

Le module définit les fonctions suivantes :

`traceback.print_tb(tb, limit=None, file=None)`

Print up to *limit* stack trace entries from traceback object *tb* (starting from the caller's frame) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. If *file* is omitted or `None`, the output goes to `sys.stderr`; otherwise it should be an open file or file-like object to receive the output.

Modifié dans la version 3.5 : Added negative *limit* support.

`traceback.print_exception(etype, value, tb, limit=None, file=None, chain=True)`

Print exception information and stack trace entries from traceback object *tb* to *file*. This differs from `print_tb()` in the following ways :

- if *tb* is not `None`, it prints a header `Traceback (most recent call last):`
- it prints the exception *etype* and *value* after the stack trace
- if `type(value)` is `SyntaxError` and *value* has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

The optional *limit* argument has the same meaning as for `print_tb()`. If *chain* is true (the default), then chained exceptions (the `__cause__` or `__context__` attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

Modifié dans la version 3.5 : The *etype* argument is ignored and inferred from the type of *value*.

`traceback.print_exc(limit=None, file=None, chain=True)`

This is a shorthand for `print_exception(*sys.exc_info(), limit, file, chain)`.

`traceback.print_last(limit=None, file=None, chain=True)`

This is a shorthand for `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file, chain)`. In general it will work only after an exception has reached an interactive prompt (see `sys.last_type`).

`traceback.print_stack(f=None, limit=None, file=None)`

Print up to *limit* stack trace entries (starting from the invocation point) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *file* argument has the same meaning as for `print_tb()`.

Modifié dans la version 3.5 : Added negative *limit* support.

`traceback.extract_tb(tb, limit=None)`

Return a *StackSummary* object representing a list of « pre-processed » stack trace entries extracted from the traceback object *tb*. It is useful for alternate formatting of stack traces. The optional *limit* argument has the same meaning as for *print_tb()*. A « pre-processed » stack trace entry is a *FrameSummary* object containing attributes *filename*, *lineno*, *name*, and *line* representing the information that is usually printed for a stack trace. The *line* is a string with leading and trailing whitespace stripped; if the source is not available it is *None*.

`traceback.extract_stack(f=None, limit=None)`

Extract the raw traceback from the current stack frame. The return value has the same format as for *extract_tb()*. The optional *f* and *limit* arguments have the same meaning as for *print_stack()*.

`traceback.format_list(extracted_list)`

Given a list of tuples or *FrameSummary* objects as returned by *extract_tb()* or *extract_stack()*, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not *None*.

`traceback.format_exception(etype, value)`

Format the exception part of a traceback. The arguments are the exception type and value such as given by *sys.last_type* and *sys.last_value*. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for *SyntaxError* exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

`traceback.format_exception(etype, value, tb, limit=None, chain=True)`

Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to *print_exception()*. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does *print_exception()*.

Modifié dans la version 3.5 : The *etype* argument is ignored and inferred from the type of *value*.

`traceback.format_exc(limit=None, chain=True)`

This is like *print_exc(limit)* but returns a string instead of printing to a file.

`traceback.format_tb(tb, limit=None)`

A shorthand for *format_list(extract_tb(tb, limit))*.

`traceback.format_stack(f=None, limit=None)`

A shorthand for *format_list(extract_stack(f, limit))*.

`traceback.clear_frames(tb)`

Clears the local variables of all the stack frames in a traceback *tb* by calling the *clear()* method of each frame object.

Nouveau dans la version 3.4.

`traceback.walk_stack(f)`

Walk a stack following *f.f_back* from the given frame, yielding the frame and line number for each frame. If *f* is *None*, the current stack is used. This helper is used with *StackSummary.extract()*.

Nouveau dans la version 3.5.

`traceback.walk_tb(tb)`

Walk a traceback following *tb_next* yielding the frame and line number for each frame. This helper is used with *StackSummary.extract()*.

Nouveau dans la version 3.5.

The module also defines the following classes :

29.9.1 `TracebackException` Objects

Nouveau dans la version 3.5.

`TracebackException` objects are created from actual exceptions to capture data for later printing in a lightweight fashion.

class `traceback.TracebackException` (*exc_type, exc_value, exc_traceback, *, limit=None, lookup_lines=True, capture_locals=False*)

Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the `StackSummary` class.

Note that when locals are captured, they are also shown in the traceback.

__cause__

A `TracebackException` of the original `__cause__`.

__context__

A `TracebackException` of the original `__context__`.

__suppress_context__

The `__suppress_context__` value from the original exception.

stack

A `StackSummary` representing the traceback.

exc_type

The class of the original traceback.

filename

For syntax errors - the file name where the error occurred.

lineno

For syntax errors - the line number where the error occurred.

text

For syntax errors - the text where the error occurred.

offset

For syntax errors - the offset into the text where the error occurred.

msg

For syntax errors - the compiler error message.

classmethod `from_exception` (*exc, *, limit=None, lookup_lines=True, capture_locals=False*)

Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the `StackSummary` class.

Note that when locals are captured, they are also shown in the traceback.

format (**, chain=True*)

Format the exception.

If *chain* is not `True`, `__cause__` and `__context__` will not be formatted.

The return value is a generator of strings, each ending in a newline and some containing internal newlines.

`print_exception()` is a wrapper around this method which just prints the lines to a file.

The message indicating which exception occurred is always the last string in the output.

format_exception_only ()

Format the exception part of the traceback.

The return value is a generator of strings, each ending in a newline.

Normally, the generator emits a single string; however, for `SyntaxError` exceptions, it emits several lines that (when printed) display detailed information about where the syntax error occurred.

The message indicating which exception occurred is always the last string in the output.

29.9.2 StackSummary Objects

Nouveau dans la version 3.5.

StackSummary objects represent a call stack ready for formatting.

class `traceback.StackSummary`

classmethod `extract` (*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Construct a *StackSummary* object from a frame generator (such as is returned by *walk_stack()* or *walk_tb()*).

If *limit* is supplied, only this many frames are taken from *frame_gen*. If *lookup_lines* is `False`, the returned *FrameSummary* objects will not have read their lines in yet, making the cost of creating the *StackSummary* cheaper (which may be valuable if it may not actually get formatted). If *capture_locals* is `True` the local variables in each *FrameSummary* are captured as object representations.

classmethod `from_list` (*a_list*)

Construct a *StackSummary* object from a supplied list of *FrameSummary* objects or old-style list of tuples. Each tuple should be a 4-tuple with filename, lineno, name, line as the elements.

format ()

Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.

For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

Modifié dans la version 3.6 : Long sequences of repeated frames are now abbreviated.

29.9.3 FrameSummary Objects

Nouveau dans la version 3.5.

FrameSummary objects represent a single frame in a traceback.

class `traceback.FrameSummary` (*filename*, *lineno*, *name*, *lookup_line=True*, *locals=None*, *line=None*)

Represent a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frames locals included in it. If *lookup_line* is `False`, the source code is not looked up until the *FrameSummary* has the *line* attribute accessed (which also happens when casting it to a tuple). *line* may be directly provided, and will prevent line lookups happening at all. *locals* is an optional local variable dictionary, and if supplied the variable representations are stored in the summary for later display.

29.9.4 Traceback Examples

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the *code* module.

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
```

(suite sur la page suivante)

(suite de la page précédente)

```

        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)

```

The following example demonstrates the different ways to print and format the exception and traceback :

```

import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("*** print_tb:")
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print("*** print_exception:")
    # exc_type below is ignored on 3.5 and later
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                             limit=2, file=sys.stdout)

    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    # exc_type below is ignored on 3.5 and later
    print(repr(traceback.format_exception(exc_type, exc_value,
                                         exc_traceback)))

    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc_traceback)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc_traceback)))
    print("*** tb_lineno:", exc_traceback.tb_lineno)

```

The output for the example would look similar to this :

```

*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):

```

(suite sur la page suivante)

(suite de la page précédente)

```

File "<doctest...>", line 10, in <module>
    lumberjack()
File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_death>]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n']
*** tb_lineno: 10

```

The following example shows the different ways to print and format the stack :

```

>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
File "<doctest>", line 10, in <module>
    another_function()
File "<doctest>", line 3, in another_function
    lumberstack()
File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_
↳ stack()))\n']

```

This last example demonstrates the final few formatting functions :

```

>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                       ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']

```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range\n']
```

29.10 `__future__` — Définitions des futurs

Source code : `Lib/_future_.py`

Le module `__future__` est un vrai module, et il a trois objectifs :

- éviter de dérouter les outils existants qui analysent les instructions d'importation et s'attendent à trouver les modules qu'ils importent ;
- s'assurer que les instructions `*future*` lancées sous les versions antérieures à 2.1 lèvent au moins des exceptions à l'exécution (l'importation du module `__future__` échoue, car il n'y avait pas de module de ce nom avant 2.1) ;
- Pour documenter le phasage de changements entraînant des incompatibilités : introduction, utilisation obligatoire. Il s'agit d'une forme de documentation exécutable, qui peut être inspectée par un programme en important `__future__` et en examinant son contenu.

Chaque instruction dans `__future__.py` est de la forme :

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                        CompilerFlag)
```

où, normalement, *OptionalRelease* est inférieur à *MandatoryRelease*, et les deux sont des quintuplets de la même forme que `sys.version_info` :

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

OptionalRelease enregistre la première version dans laquelle la fonctionnalité a été acceptée.

Dans le cas d'un *MandatoryRelease* qui n'a pas encore eu lieu, *MandatoryRelease* prédit la *release* dans laquelle la fonctionnalité deviendra un élément du langage.

Sinon *MandatoryRelease* enregistre lorsque la fonctionnalité est devenue une partie du langage ; dans cette version ou les suivantes, les modules n'ont plus besoin d'une déclaration *future* pour utiliser la fonctionnalité en question, mais ils peuvent continuer à utiliser ces importations.

MandatoryRelease peut également être `None`, ce qui signifie qu'une fonction planifiée a été abandonnée.

Les instances de classe `_Feature` ont deux méthodes correspondantes, `getOptionalRelease()` et `getMandatoryRelease()`.

CompilerFlag est un drapeau (chaque bit représente un champ) qui doit être passé en tant que quatrième argument à la fonction native `compile()` pour activer la fonctionnalité dans le code compilé dynamiquement. Cet indicateur est stocké dans l'attribut `compiler_flag` dans les instances de `_Feature`.

Aucune fonctionnalité ne sera jamais supprimée de `__future__`. Depuis son introduction dans Python 2.1, les fonctionnalités suivantes ont trouvé leur places dans le langage utilisant ce mécanisme :

fonctionnalité	optionnel dans	obligatoire dans	effet
nested_scopes	2.1.0b1	2.2	PEP 227 : <i>Portées imbriquées</i>
générateurs	2.2.0a1	2.3	PEP 255 : <i>Générateurs simples</i>
division	2.2.0a2	3.0	PEP 238 : <i>Changement de l'opérateur de division</i>
absolute_import	2.5.0a1	3.0	PEP 328 : <i>Importations : multilignes et absolues/relatives</i> (resource en anglais)
with_statement	2.5.0a1	2.6	PEP 343 : <i>L'instruction « with »</i>
print_function	2.6.0a2	3.0	PEP 3105 : <i>Transformation de print en fonction</i>
unicode_literals	2.6.0a2	3.0	PEP 3112 : <i>Chaînes d'octets littéraux en Python 3000</i>
generator_stop	3.5.0b1	3.7	PEP 479 : <i>Gestion de *StopIteration à l'intérieur des générateurs*</i>

Voir aussi :

future Comment le compilateur gère les importations « futures ».

29.11 gc — Garbage Collector interface

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling `gc.disable()`. To debug a leaking program call `gc.set_debug(gc.DEBUG_LEAK)`. Notice that this includes `gc.DEBUG_SAVEALL`, causing garbage-collected objects to be saved in `gc.garbage` for inspection.

The `gc` module provides the following functions :

`gc.enable()`

Enable automatic garbage collection.

`gc.disable()`

Disable automatic garbage collection.

`gc.isenabled()`

Returns true if automatic collection is enabled.

`gc.collect(generation=2)`

With no arguments, run a full collection. The optional argument *generation* may be an integer specifying which generation to collect (from 0 to 2). A *ValueError* is raised if the generation number is invalid. The number of unreachable objects found is returned.

The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular *float*.

`gc.set_debug(flags)`

Set the garbage collection debugging flags. Debugging information will be written to `sys.stderr`. See below for a list of debugging flags which can be combined using bit operations to control debugging.

`gc.get_debug()`

Return the debugging flags currently set.

`gc.get_objects()`

Returns a list of all objects tracked by the collector, excluding the list returned.

`gc.get_stats()`

Return a list of three per-generation dictionaries containing collection statistics since interpreter start. The number of keys may change in the future, but currently each dictionary will contain the following items :

- `collections` is the number of times this generation was collected ;
- `collected` is the total number of objects collected inside this generation ;
- `uncollectable` is the total number of objects which were found to be uncollectable (and were therefore moved to the *garbage* list) inside this generation.

Nouveau dans la version 3.4.

`gc.set_threshold(threshold0[, threshold1[, threshold2]])`

Set the garbage collection thresholds (the collection frequency). Setting *threshold0* to zero disables collection.

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation 0). If an object survives a collection it is moved into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than *threshold1* times since generation 1 has been examined, then generation 1 is examined as well. Similarly, *threshold2* controls the number of collections of generation 1 before collecting generation 2.

`gc.get_count()`

Return the current collection counts as a tuple of (*count0*, *count1*, *count2*).

`gc.get_threshold()`

Return the current collection thresholds as a tuple of (*threshold0*, *threshold1*, *threshold2*).

`gc.get_referrers(*objs)`

Return the list of objects that directly refer to any of *objs*. This function will only locate those containers which support garbage collection ; extension types which do refer to other objects but do not support garbage collection will not be found.

Note that objects which have already been dereferenced, but which live in cycles and have not yet been collected by the garbage collector can be listed among the resulting referrers. To get only currently live objects, call *collect()* before calling *get_referrers()*.

Care must be taken when using objects returned by *get_referrers()* because some of them could still be under construction and hence in a temporarily invalid state. Avoid using *get_referrers()* for any purpose other than debugging.

`gc.get_referents(*objs)`

Return a list of objects directly referred to by any of the arguments. The referents returned are those objects visited by the arguments' C-level *tp_traverse* methods (if any), and may not be all objects actually directly reachable. *tp_traverse* methods are supported only by objects that support garbage collection, and are only required to visit objects that may be involved in a cycle. So, for example, if an integer is directly reachable from an argument, that integer object may or may not appear in the result list.

`gc.is_tracked(obj)`

Returns *True* if the object is currently tracked by the garbage collector, *False* otherwise. As a general rule, instances of atomic types aren't tracked and instances of non-atomic types (containers, user-defined objects...) are. However, some type-specific optimizations can be present in order to suppress the garbage collector footprint of simple instances (e.g. dicts containing only atomic keys and values) :

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

Nouveau dans la version 3.1.

The following variables are provided for read-only access (you can mutate the values but should not rebind them) :

`gc.garbage`

A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). Starting with Python 3.4, this list should be empty most of the time, except when using instances of C extension types with a non-NULL `tp_del` slot.

If `DEBUG_SAVEALL` is set, then all unreachable objects will be added to this list rather than freed.

Modifié dans la version 3.2 : If this list is non-empty at *interpreter shutdown*, a *ResourceWarning* is emitted, which is silent by default. If `DEBUG_UNCOLLECTABLE` is set, in addition all uncollectable objects are printed.

Modifié dans la version 3.4 : Following **PEP 442**, objects with a `__del__()` method don't end up in *gc.garbage* anymore.

`gc.callbacks`

A list of callbacks that will be invoked by the garbage collector before and after collection. The callbacks will be called with two arguments, *phase* and *info*.

phase can be one of two values :

- « start » : The garbage collection is about to start.
- « stop » : The garbage collection has finished.

info is a dict providing more information for the callback. The following keys are currently defined :

- « generation » : The oldest generation being collected.
- « collected » : When *phase* is « stop », the number of objects successfully collected.
- « uncollectable » : When *phase* is « stop », the number of objects that could not be collected and were put in *gc.garbage*.

Applications can add their own callbacks to this list. The primary use cases are :

Gathering statistics about garbage collection, such as how often various generations are collected, and how long the collection takes.

Allowing applications to identify and clear their own uncollectable types when they appear in *gc.garbage*.

Nouveau dans la version 3.3.

The following constants are provided for use with `set_debug()` :

`gc.DEBUG_STATS`

Print statistics during collection. This information can be useful when tuning the collection frequency.

`gc.DEBUG_COLLECTABLE`

Print information on collectable objects found.

`gc.DEBUG_UNCOLLECTABLE`

Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the *gc.garbage* list.

Modifié dans la version 3.2 : Also print the contents of the *gc.garbage* list at *interpreter shutdown*, if it isn't empty.

gc.DEBUG_SAVEALL

When set, all unreachable objects found will be appended to *garbage* rather than being freed. This can be useful for debugging a leaking program.

gc.DEBUG_LEAK

The debugging flags necessary for the collector to print information about a leaking program (equal to `DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL`).

29.12 inspect — Inspect live objects

Code source : <Lib/inspect.py>

The *inspect* module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module : type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

29.12.1 Types and members

The *getmembers()* function retrieves the members of an object such as a class or module. The functions whose names begin with « is » are mainly provided as convenient choices for the second argument to *getmembers()*. They also help you determine when you can expect to find the following special attributes :

Type	Attribut	Description
module	<code>__doc__</code>	documentation string
	<code>__file__</code>	filename (missing for built-in modules)
	<code>__name__</code>	name with which this module was defined
classe	<code>__doc__</code>	documentation string
	<code>__name__</code>	name with which this class was defined
	<code>__qualname__</code>	nom qualifié
méthode	<code>__module__</code>	name of module in which this class was defined
	<code>__doc__</code>	documentation string
	<code>__name__</code>	name with which this method was defined
fonction	<code>__qualname__</code>	nom qualifié
	<code>__func__</code>	function object containing implementation of method
	<code>__self__</code>	instance to which this method is bound, or <code>None</code>
fonction	<code>__doc__</code>	documentation string
	<code>__name__</code>	name with which this function was defined
	<code>__qualname__</code>	nom qualifié
fonction	<code>__code__</code>	code object containing compiled function <i>bytecode</i>
	<code>__defaults__</code>	tuple of any default values for positional or keyword parameters
	<code>__kwdefaults__</code>	mapping of any default values for keyword-only parameters
fonction	<code>__globals__</code>	global namespace in which this function was defined
	<code>__annotations__</code>	mapping of parameters names to annotations; "return" key is reserved for return annotations.
traceback	<code>tb_frame</code>	frame object at this level
	<code>tb_lasti</code>	index of last attempted instruction in bytecode
	<code>tb_lineno</code>	current line number in Python source code

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Type	Attribut	Description
frame	tb_next	next inner traceback object (called by this level)
	f_back	next outer frame object (this frame's caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
code	f_locals	local namespace seen by this frame
	f_trace	tracing function for this frame, or None
	co_argcount	number of arguments (not including keyword only arguments, * or ** args)
	co_code	string of raw compiled bytecode
	co_cellvars	tuple of names of cell variables (referenced by containing scopes)
	co_consts	tuple of constants used in the bytecode
	co_filename	name of file in which this code object was created
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap of CO_* flags, read more here
	co_lnotab	encoded mapping of line numbers to bytecode indices
	co_freevars	tuple of names of free variables (referenced via a function's closure)
	co_kwonlyargcount	number of keyword only arguments (not including ** arg)
	co_name	name with which this code object was defined
	co_names	tuple of names of local variables
	co_nlocals	number of local variables
	co_stacksize	virtual machine stack space required
	co_varnames	tuple of names of arguments and local variables
générateur	__name__	name
	__qualname__	nom qualifié
	gi_frame	frame
	gi_running	is the generator running?
	gi_code	code
coroutine	gi_yieldfrom	object being iterated by <code>yield from</code> , or None
	__name__	name
	__qualname__	nom qualifié
	cr_await	object being awaited on, or None
	cr_frame	frame
	cr_running	is the coroutine running?
builtin	cr_code	code
	__doc__	documentation string
	__name__	original name of this function or method
	__qualname__	nom qualifié
	__self__	instance to which a method is bound, or None

Modifié dans la version 3.5 : Add `__qualname__` and `gi_yieldfrom` attributes to generators.

The `__name__` attribute of generators is now set from the function name, instead of the code name, and it can now be modified.

`inspect.getmembers(object[, predicate])`

Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional *predicate* argument is supplied, only members for which the predicate returns a true value are included.

Note : `getmembers()` will only return class attributes defined in the metaclass when the argument is a class

and those attributes have been listed in the metaclass” custom `__dir__()`.

`inspect.getmodule`**name** (*path*)

Return the name of the module named by the file *path*, without including the names of enclosing packages. The file extension is checked against all of the entries in `importlib.machinery.all_suffixes()`. If it matches, the final path component is returned with the extension removed. Otherwise, `None` is returned.

Note that this function *only* returns a meaningful name for actual Python modules - paths that potentially refer to Python packages will still return `None`.

Modifié dans la version 3.3 : The function is based directly on `importlib`.

`inspect.ismodule` (*object*)

Return true if the object is a module.

`inspect.isclass` (*object*)

Return true if the object is a class, whether built-in or created in Python code.

`inspect.ismethod` (*object*)

Return true if the object is a bound method written in Python.

`inspect.isfunction` (*object*)

Return true if the object is a Python function, which includes functions created by a *lambda* expression.

`inspect.isgeneratorfunction` (*object*)

Return true if the object is a Python generator function.

`inspect.isgenerator` (*object*)

Return true if the object is a generator.

`inspect.iscoroutinefunction` (*object*)

Return true if the object is a *coroutine function* (a function defined with an `async def` syntax).

Nouveau dans la version 3.5.

`inspect.iscoroutine` (*object*)

Return true if the object is a *coroutine* created by an `async def` function.

Nouveau dans la version 3.5.

`inspect.isawaitable` (*object*)

Return true if the object can be used in `await` expression.

Can also be used to distinguish generator-based coroutines from regular generators :

```
def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

Nouveau dans la version 3.5.

`inspect.isasyncgenfunction` (*object*)

Return true if the object is an *asynchronous generator* function, for example :

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

Nouveau dans la version 3.6.

`inspect.isasyncgen(object)`

Return true if the object is an *asynchronous generator iterator* created by an *asynchronous generator* function.

Nouveau dans la version 3.6.

`inspect.istraceback(object)`

Return true if the object is a traceback.

`inspect.isframe(object)`

Return true if the object is a frame.

`inspect.iscode(object)`

Return true if the object is a code.

`inspect.isbuiltin(object)`

Return true if the object is a built-in function or a bound built-in method.

`inspect.isroutine(object)`

Return true if the object is a user-defined or built-in function or method.

`inspect.isabstract(object)`

Return true if the object is an abstract base class.

`inspect.ismethoddescriptor(object)`

Return true if the object is a method descriptor, but not if `ismethod()`, `isclass()`, `isfunction()` or `isbuiltin()` are true.

This, for example, is true of `int.__add__`. An object passing this test has a `__get__()` method but not a `__set__()` method, but beyond that the set of attributes varies. A `__name__` attribute is usually sensible, and `__doc__` often is.

Methods implemented via descriptors that also pass one of the other tests return false from the `ismethoddescriptor()` test, simply because the other tests promise more – you can, e.g., count on having the `__func__` attribute (etc) when an object passes `ismethod()`.

`inspect.isdatadescriptor(object)`

Return true if the object is a data descriptor.

Data descriptors have both a `__get__` and a `__set__` method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have `__name__` and `__doc__` attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

`inspect.isgetsetdescriptor(object)`

Return true if the object is a getset descriptor.

CPython implementation detail : getsets are attributes defined in extension modules via `PyGetSetDef` structures. For Python implementations without such types, this method will always return `False`.

`inspect.ismemberdescriptor(object)`

Return true if the object is a member descriptor.

CPython implementation detail : Member descriptors are attributes defined in extension modules via `PyMemberDef` structures. For Python implementations without such types, this method will always return `False`.

29.12.2 Retrieving source code

`inspect.getdoc(object)`

Get the documentation string for an object, cleaned up with `cleandoc()`. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy.

Modifié dans la version 3.5 : Documentation strings are now inherited if not overridden.

`inspect.getcomments(object)`

Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module). If the object's source code is unavailable, return `None`. This could happen if the object has been defined in C or the interactive shell.

`inspect.getfile(object)`

Return the name of the (text or binary) file in which an object was defined. This will fail with a `TypeError` if the object is a built-in module, class, or function.

`inspect.getmodule(object)`

Try to guess which module an object was defined in.

`inspect.getsourcefile(object)`

Return the name of the Python source file in which an object was defined. This will fail with a `TypeError` if the object is a built-in module, class, or function.

`inspect.getsourcelines(object)`

Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An `OSError` is raised if the source code cannot be retrieved.

Modifié dans la version 3.3 : `OSError` is raised instead of `IOError`, now an alias of the former.

`inspect.getsource(object)`

Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An `OSError` is raised if the source code cannot be retrieved.

Modifié dans la version 3.3 : `OSError` is raised instead of `IOError`, now an alias of the former.

`inspect.cleandoc(doc)`

Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

29.12.3 Introspecting callables with the Signature object

Nouveau dans la version 3.3.

The Signature object represents the call signature of a callable object and its return annotation. To retrieve a Signature object, use the `signature()` function.

`inspect.signature(callable, *, follow_wrapped=True)`

Return a `Signature` object for the given callable :

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass
```

(suite sur la page suivante)

(suite de la page précédente)

```

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b:int, **kwargs)'

>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>

```

Accepts a wide range of python callables, from plain functions and classes to `functools.partial()` objects. Raises `ValueError` if no signature can be provided, and `TypeError` if that type of object is not supported. Nouveau dans la version 3.5 : `follow_wrapped` parameter. Pass `False` to get a signature of callable specifically (`callable.__wrapped__` will not be used to unwrap decorated callables.)

Note : Some callables may not be introspectable in certain implementations of Python. For example, in CPython, some built-in functions defined in C provide no metadata about their arguments.

class `inspect.Signature` (*parameters=None, *, return_annotation=Signature.empty*)

A Signature object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a `Parameter` object in its `parameters` collection.

The optional `parameters` argument is a sequence of `Parameter` objects, which is validated to check that there are no parameters with duplicate names, and that the parameters are in the right order, i.e. positional-only first, then positional-or-keyword, and that parameters with defaults follow parameters without defaults.

The optional `return_annotation` argument, can be an arbitrary Python object, is the « return » annotation of the callable.

Signature objects are *immutable*. Use `Signature.replace()` to make a modified copy.

Modifié dans la version 3.5 : Signature objects are picklable and hashable.

empty

A special class-level marker to specify absence of a return annotation.

parameters

An ordered mapping of parameters' names to the corresponding `Parameter` objects.

return_annotation

The « return » annotation for the callable. If the callable has no « return » annotation, this attribute is set to `Signature.empty`.

bind (**args, **kwargs*)

Create a mapping from positional and keyword arguments to parameters. Returns `BoundArguments` if `*args` and `**kwargs` match the signature, or raises a `TypeError`.

bind_partial (**args, **kwargs*)

Works the same way as `Signature.bind()`, but allows the omission of some required arguments (mimics `functools.partial()` behavior.) Returns `BoundArguments`, or raises a `TypeError` if the passed arguments do not match the signature.

replace (**, parameters[], return_annotation[]*)

Create a new Signature instance based on the instance `replace` was invoked on. It is possible to pass different parameters and/or return_annotation to override the corresponding properties of the base signature. To remove return_annotation from the copied Signature, pass in `Signature.empty`.

```

>>> def test(a, b):
...     pass
>>> sig = signature(test)

```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

classmethod `from_callable(obj, *, follow_wrapped=True)`

Return a *Signature* (or its subclass) object for a given callable `obj`. Pass `follow_wrapped=False` to get a signature of `obj` without unwrapping its `__wrapped__` chain.

This method simplifies subclassing of *Signature*:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(min)
assert isinstance(sig, MySignature)
```

Nouveau dans la version 3.5.

class `inspect.Parameter(name, kind, *, default=Parameter.empty, annotation=Parameter.empty)`

Parameter objects are *immutable*. Instead of modifying a *Parameter* object, you can use *Parameter.replace()* to create a modified copy.

Modifié dans la version 3.5 : Parameter objects are picklable and hashable.

empty

A special class-level marker to specify absence of default values and annotations.

name

The name of the parameter as a string. The name must be a valid Python identifier.

CPython implementation detail : CPython generates implicit parameter names of the form `.0` on the code objects used to implement comprehensions and generator expressions.

Modifié dans la version 3.6 : These parameter names are exposed by this module as names like `implicit0`.

default

The default value for the parameter. If the parameter has no default value, this attribute is set to *Parameter.empty*.

annotation

The annotation for the parameter. If the parameter has no annotation, this attribute is set to *Parameter.empty*.

kind

Describes how argument values are bound to the parameter. Possible values (accessible via *Parameter*, like *Parameter.KEYWORD_ONLY*):

Nom	Signification
<i>POSITIONAL_ONLY</i>	Value must be supplied as a positional argument. Python has no explicit syntax for defining positional-only parameters, but many built-in and extension module functions (especially those that accept only one or two parameters) accept them.
<i>POSITIONAL_OR_KEYWORD</i>	Value may be supplied as either a keyword or positional argument (this is the standard binding behaviour for functions implemented in Python.)
<i>VAR_POSITIONAL</i>	A tuple of positional arguments that aren't bound to any other parameter. This corresponds to a <code>*args</code> parameter in a Python function definition.
<i>KEYWORD_ONLY</i>	Value must be supplied as a keyword argument. Keyword only parameters are those which appear after a <code>*</code> or <code>*args</code> entry in a Python function definition.
<i>VAR_KEYWORD</i>	A dict of keyword arguments that aren't bound to any other parameter. This corresponds to a <code>**kwargs</code> parameter in a Python function definition.

Example : print all keyword-only arguments without default values :

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...         param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

replace (*[, name][, kind][, default][, annotation])

Create a new `Parameter` instance based on the instance replaced was invoked on. To override a `Parameter` attribute, pass the corresponding argument. To remove a default value or/and an annotation from a `Parameter`, pass `Parameter.empty`.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
'foo: 'spam''
```

Modifié dans la version 3.4 : In Python 3.3 `Parameter` objects were allowed to have `name` set to `None` if their `kind` was set to `POSITIONAL_ONLY`. This is no longer permitted.

class `inspect.BoundArguments`

Result of a `Signature.bind()` or `Signature.bind_partial()` call. Holds the mapping of arguments to the function's parameters.

arguments

An ordered, mutable mapping (`collections.OrderedDict`) of parameters' names to arguments' values. Contains only explicitly bound arguments. Changes in `arguments` will reflect in `args` and `kwargs`. Should be used in conjunction with `Signature.parameters` for any argument processing purposes.

Note : Arguments for which `Signature.bind()` or `Signature.bind_partial()` relied on a default value are skipped. However, if needed, use `BoundArguments.apply_defaults()` to add them.

args

A tuple of positional arguments values. Dynamically computed from the `arguments` attribute.

kwargs

A dict of keyword arguments values. Dynamically computed from the `arguments` attribute.

signature

A reference to the parent `Signature` object.

apply_defaults()

Set default values for missing arguments.

For variable-positional arguments (`*args`) the default is an empty tuple.

For variable-keyword arguments (`**kwargs`) the default is an empty dict.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
OrderedDict([('a', 'spam'), ('b', 'ham'), ('args', ())])
```

Nouveau dans la version 3.5.

The `args` and `kwargs` properties can be used to invoke functions :

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

Voir aussi :

PEP 362 - Function Signature Object. The detailed specification, implementation details and examples.

29.12.4 Classes et fonctions

`inspect.getclasstree(classes, unique=False)`

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the *unique* argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

`inspect.getargspec(func)`

Get the names and default values of a Python function's parameters. A *named tuple* `ArgSpec(args, varargs, keywords, defaults)` is returned. *args* is a list of the parameter names. *varargs* and *keywords* are the names of the `*` and `**` parameters or `None`. *defaults* is a tuple of default argument values or `None` if there are no default arguments; if this tuple has *n* elements, they correspond to the last *n* elements listed in *args*.

Obsolète depuis la version 3.0 : Use `getfullargspec()` for an updated API that is usually a drop-in replacement, but also correctly handles function annotations and keyword-only parameters.

Alternatively, use `signature()` and *Signature Object*, which provide a more structured introspection API for callables.

`inspect.getfullargspec(func)`

Get the names and default values of a Python function's parameters. A *named tuple* is returned :

`FullArgSpec(args, varargs, varkw, defaults, kwoonlyargs, kwoonlydefaults, annotations)`

args is a list of the positional parameter names. *varargs* is the name of the `*` parameter or `None` if arbitrary positional arguments are not accepted. *varkw* is the name of the `**` parameter or `None` if arbitrary keyword arguments are not accepted. *defaults* is an *n*-tuple of default argument values corresponding to the last *n* positional parameters, or `None` if there are no such defaults defined. *kwoonlyargs* is a list of keyword-only parameter names. *kwoonlydefaults* is a dictionary mapping parameter names from *kwoonlyargs* to the default values used if no argument is supplied. *annotations* is a dictionary mapping parameter names to annotations. The special key `"return"` is used to report the function return value annotation (if any).

Note that `signature()` and *Signature Object* provide the recommended API for callable introspection, and support additional behaviours (like positional-only arguments) that are sometimes encountered in extension module APIs. This function is retained primarily for use in code that needs to maintain compatibility with the Python 2 `inspect` module API.

Modifié dans la version 3.4 : This function is now based on `signature()`, but still ignores `__wrapped__` attributes and includes the already bound first parameter in the signature output for bound methods.

Modifié dans la version 3.6 : This method was previously documented as deprecated in favour of `signature()` in Python 3.5, but that decision has been reversed in order to restore a clearly supported standard interface for single-source Python 2/3 code migrating away from the legacy `getargspec()` API.

`inspect.getargvalues(frame)`

Get information about arguments passed into a particular frame. A *named tuple* `ArgInfo(args, varargs, keywords, locals)` is returned. `args` is a list of the argument names. `varargs` and `keywords` are the names of the `*` and `**` arguments or `None`. `locals` is the locals dictionary of the given frame.

Note : This function was inadvertently marked as deprecated in Python 3.5.

`inspect.formatargspec(args[, varargs, varkw, defaults, kwoonlyargs, kwoonlydefaults, annotations[, formatarg, formatvarargs, formatvarkw, formatvalue, formatreturns, formatannotations]])`

Format a pretty argument spec from the values returned by `getfullargspec()`.

The first seven arguments are (`args`, `varargs`, `varkw`, `defaults`, `kwoonlyargs`, `kwoonlydefaults`, `annotations`).

The other six arguments are functions that are called to turn argument names, `*` argument name, `**` argument name, default values, return annotation and individual annotations into strings, respectively.

Par exemple :

```
>>> from inspect import formatargspec, getfullargspec
>>> def f(a: int, b: float):
...     pass
...
>>> formatargspec(*getfullargspec(f))
'(a: int, b: float)'
```

Obsolète depuis la version 3.5 : Use `signature()` and *Signature Object*, which provide a better introspecting API for callables.

`inspect.formatargvalues(args[, varargs, varkw, locals, formatarg, formatvarargs, formatvarkw, formatvalue])`

Format a pretty argument spec from the four values returned by `getargvalues()`. The `format*` arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

Note : This function was inadvertently marked as deprecated in Python 3.5.

`inspect.getmro(cls)`

Return a tuple of class `cls`'s base classes, including `cls`, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on `cls`'s type. Unless a very peculiar user-defined metatype is in use, `cls` will be the first element of the tuple.

`inspect.getcallargs(func, *args, **kwds)`

Bind the `args` and `kwds` to the argument names of the Python function or method `func`, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned, mapping the argument names (including the names of the `*` and `**` arguments, if any) to their values from `args` and `kwds`. In case of invoking `func` incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example :

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
```

(suite sur la page suivante)

(suite de la page précédente)

```

...     pass
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'

```

Nouveau dans la version 3.2.

Obsolète depuis la version 3.5 : Use `Signature.bind()` and `Signature.bind_partial()` instead.

`inspect.getclosurevars(func)`

Get the mapping of external name references in a Python function or method *func* to their current values. A *named tuple* `ClosureVars(nonlocals, globals, builtins, unbound)` is returned. *nonlocals* maps referenced names to lexical closure variables, *globals* to the function's module globals and *builtins* to the builtins visible from the function body. *unbound* is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

`TypeError` is raised if *func* is not a Python function or method.

Nouveau dans la version 3.3.

`inspect.unwrap(func, *, stop=None)`

Get the object wrapped by *func*. It follows the chain of `__wrapped__` attributes returning the last object in the chain.

stop is an optional callback accepting an object in the wrapper chain as its sole argument that allows the unwrapping to be terminated early if the callback returns a true value. If the callback never returns a true value, the last object in the chain is returned as usual. For example, `signature()` uses this to stop unwrapping if any object in the chain has a `__signature__` attribute defined.

`ValueError` is raised if a cycle is encountered.

Nouveau dans la version 3.4.

29.12.5 The interpreter stack

When the following functions return « frame records, » each record is a *named tuple* `FrameInfo(frame, filename, lineno, function, code_context, index)`. The tuple contains the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

Modifié dans la version 3.5 : Return a named tuple instead of a tuple.

Note : Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using `gc.disable()`. For example :

```

def handle_stackframe_without_leak():
    frame = inspect.currentframe()

```

(suite sur la page suivante)


```
try:
    # do something with the frame
finally:
    del frame
```

If you want to keep the frame around (for example to print a traceback later), you can also break reference cycles by using the `frame.clear()` method.

The optional *context* argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

`inspect.getframeinfo(frame, context=1)`

Get information about a frame or traceback object. A *named tuple* `Traceback(filename, lineno, function, code_context, index)` is returned.

`inspect.getouterframes(frame, context=1)`

Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of *frame*. The first entry in the returned list represents *frame*; the last entry represents the outermost call on *frame*'s stack.

Modifié dans la version 3.5 : A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.getinnerframes(traceback, context=1)`

Get a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of *frame*. The first entry in the list represents *traceback*; the last entry represents where the exception was raised.

Modifié dans la version 3.5 : A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.currentframe()`

Return the frame object for the caller's stack frame.

CPython implementation detail : This function relies on Python stack frame support in the interpreter, which isn't guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns `None`.

`inspect.stack(context=1)`

Return a list of frame records for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

Modifié dans la version 3.5 : A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.trace(context=1)`

Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

Modifié dans la version 3.5 : A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

29.12.6 Fetching attributes statically

Both `getattr()` and `hasattr()` can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()` and `__getattribute__()` may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. `getattr_static()` has the same signature as `getattr()` but avoids executing code when it fetches attributes.

`inspect.getattr_static(obj, attr, default=None)`

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`.

Note : this function may not be able to retrieve all attributes that `getattr` can fetch (like dynamically created attributes) and may find attributes that `getattr` can't (like descriptors that raise `AttributeError`). It can also return descriptors objects instead of instance members.

If the instance `__dict__` is shadowed by another member (for example a property) then this function will be unable to find instance members.

Nouveau dans la version 3.2.

`getattr_static()` does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.

You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution :

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

29.12.7 Current State of Generators and Coroutines

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated. `getgeneratorstate()` allows the current state of a generator to be determined easily.

`inspect.getgeneratorstate(generator)`

Get current state of a generator-iterator.

Possible states are :

- `GEN_CREATED` : Waiting to start execution.
- `GEN_RUNNING` : Currently being executed by the interpreter.

- `GEN_SUSPENDED` : Currently suspended at a yield expression.
- `GEN_CLOSED` : Execution has completed.

Nouveau dans la version 3.2.

`inspect.getcoroutinestate` (*coroutine*)

Get current state of a coroutine object. The function is intended to be used with coroutine objects created by `async def` functions, but will accept any coroutine-like object that has `cr_running` and `cr_frame` attributes.

Possible states are :

- `CORO_CREATED` : Waiting to start execution.
- `CORO_RUNNING` : Currently being executed by the interpreter.
- `CORO_SUSPENDED` : Currently suspended at an await expression.
- `CORO_CLOSED` : Execution has completed.

Nouveau dans la version 3.5.

The current internal state of the generator can also be queried. This is mostly useful for testing purposes, to ensure that internal state is being updated as expected :

`inspect.getgeneratorlocals` (*generator*)

Get the mapping of live local variables in *generator* to their current values. A dictionary is returned that maps from variable names to values. This is the equivalent of calling `locals()` in the body of the generator, and all the same caveats apply.

If *generator* is a *generator* with no currently associated frame, then an empty dictionary is returned. `TypeError` is raised if *generator* is not a Python generator object.

CPython implementation detail : This function relies on the generator exposing a Python stack frame for introspection, which isn't guaranteed to be the case in all implementations of Python. In such cases, this function will always return an empty dictionary.

Nouveau dans la version 3.3.

`inspect.getcoroutinelocals` (*coroutine*)

This function is analogous to `getgeneratorlocals()`, but works for coroutine objects created by `async def` functions.

Nouveau dans la version 3.5.

29.12.8 Code Objects Bit Flags

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags :

`inspect.CO_OPTIMIZED`

The code object is optimized, using fast locals.

`inspect.CO_NEWLOCALS`

If set, a new dict will be created for the frame's `f_locals` when the code object is executed.

`inspect.CO_VARARGS`

The code object has a variable positional parameter (`*args`-like).

`inspect.CO_VARKEYWORDS`

The code object has a variable keyword parameter (`**kwargs`-like).

`inspect.CO_NESTED`

The flag is set when the code object is a nested function.

`inspect.CO_GENERATOR`

The flag is set when the code object is a generator function, i.e. a generator object is returned when the code object is executed.

`inspect.CO_NOFREE`

The flag is set if there are no free or cell variables.

`inspect.CO_COROUTINE`

The flag is set when the code object is a coroutine function. When the code object is executed it returns a coroutine object. See [PEP 492](#) for more details.

Nouveau dans la version 3.5.

`inspect.CO_ITERABLE_COROUTINE`

The flag is used to transform generators into generator-based coroutines. Generator objects with this flag can be used in `await` expression, and can `yield from` coroutine objects. See [PEP 492](#) for more details.

Nouveau dans la version 3.5.

`inspect.CO_ASYNC_GENERATOR`

The flag is set when the code object is an asynchronous generator function. When the code object is executed it returns an asynchronous generator object. See [PEP 525](#) for more details.

Nouveau dans la version 3.6.

Note : The flags are specific to CPython, and may not be defined in other Python implementations. Furthermore, the flags are an implementation detail, and can be removed or deprecated in future Python releases. It's recommended to use public APIs from the `inspect` module for any introspection needs.

29.12.9 Interface en ligne de commande

The `inspect` module also provides a basic introspection capability from the command line.

By default, accepts the name of a module and prints the source of that module. A class or function within the module can be printed instead by appended a colon and the qualified name of the target object.

--details

Print information about the specified object rather than the source code

29.13 `site` — Site-specific configuration hook

Code source : [Lib/site.py](#)

This module is automatically imported during initialization. The automatic import can be suppressed using the interpreter's `-S` option.

Importing this module will append site-specific paths to the module search path and add a few builtins, unless `-S` was used. In that case, this module can be safely imported with no automatic modifications to the module search path or additions to the builtins. To explicitly trigger the usual site-specific additions, call the `site.main()` function.

Modifié dans la version 3.3 : Importing the module used to trigger paths manipulation even when using `-S`.

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses `sys.prefix` and `sys.exec_prefix`; empty heads are skipped. For the tail part, it uses the empty string and then `lib/site-packages` (on Windows) or `lib/pythonX.Y/site-packages` (on Unix and Macintosh). For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to `sys.path` and also inspects the newly added path for configuration files.

Modifié dans la version 3.5 : Support for the « site-python » directory has been removed.

If a file named « `pyenv.cfg` » exists one directory above `sys.executable`, `sys.prefix` and `sys.exec_prefix` are set to that directory and it is also checked for site-packages (`sys.base_prefix` and `sys.base_exec_prefix` will always be the « real »

prefixes of the Python installation). If « `pyenv.cfg` » (a bootstrap configuration file) contains the key « `include-system-site-packages` » set to anything other than « `false` » (case-insensitive), the system-level prefixes will still also be searched for site-packages; otherwise they won't.

A path configuration file is a file whose name has the form `name.pth` and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to `sys.path`. Non-existing items are never added to `sys.path`, and no check is made that the item refers to a directory rather than a file. No item is added to `sys.path` more than once. Blank lines and lines beginning with `#` are skipped. Lines starting with `import` (followed by space or tab) are executed.

For example, suppose `sys.prefix` and `sys.exec_prefix` are set to `/usr/local`. The Python `X.Y` library is then installed in `/usr/local/lib/pythonX.Y`. Suppose this has a subdirectory `/usr/local/lib/pythonX.Y/site-packages` with three subsubdirectories, `foo`, `bar` and `spam`, and two path configuration files, `foo.pth` and `bar.pth`. Assume `foo.pth` contains the following :

```
# foo package configuration

foo
bar
bletch
```

and `bar.pth` contains :

```
# bar package configuration

bar
```

Then the following version-specific directories are added to `sys.path`, in this order :

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Note that `bletch` is omitted because it doesn't exist; the `bar` directory precedes the `foo` directory because `bar.pth` comes alphabetically before `foo.pth`; and `spam` is omitted because it is not mentioned in either path configuration file.

After these path manipulations, an attempt is made to import a module named `sitecustomize`, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the `site-packages` directory. If this import fails with an `ImportError` exception, it is silently ignored. If Python is started without output streams available, as with `pythonw.exe` on Windows (which is used by default to start IDLE), attempted output from `sitecustomize` is ignored. Any exception other than `ImportError` causes a silent and perhaps mysterious failure of the process.

After this, an attempt is made to import a module named `usercustomize`, which can perform arbitrary user-specific customizations, if `ENABLE_USER_SITE` is true. This file is intended to be created in the user `site-packages` directory (see below), which is part of `sys.path` unless disabled by `-s`. An `ImportError` will be silently ignored.

Note that for some non-Unix systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped; however the import of `sitecustomize` and `usercustomize` is still attempted.

29.13.1 Readline configuration

On systems that support *readline*, this module will also import and configure the *rlcompleter* module, if Python is started in interactive mode and without the `-S` option. The default behavior is enable tab-completion and to use `~/.python_history` as the history save file. To disable it, delete (or override) the `sys.__interactivehook__` attribute in your `sitcustomize` or `usercustomize` module or your `PYTHONSTARTUP` file.

Modifié dans la version 3.4 : Activation of *rlcompleter* and history was made automatic.

29.13.2 Module contents

`site.PREFIXES`

A list of prefixes for site-packages directories.

`site.ENABLE_USER_SITE`

Flag showing the status of the user site-packages directory. `True` means that it is enabled and was added to `sys.path`. `False` means that it was disabled by user request (with `-s` or `PYTHONNOUSERSITE`). `None` means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

`site.USER_SITE`

Path to the user site-packages for the running Python. Can be `None` if `getusersitepackages()` hasn't been called yet. Default value is `~/.local/lib/pythonX.Y/site-packages` for UNIX and non-framework Mac OS X builds, `~/Library/Python/X.Y/lib/python/site-packages` for Mac framework builds, and `%APPDATA%\Python\PythonXY\site-packages` on Windows. This directory is a site directory, which means that `.pth` files in it will be processed.

`site.USER_BASE`

Path to the base directory for the user site-packages. Can be `None` if `getuserbase()` hasn't been called yet. Default value is `~/.local` for UNIX and Mac OS X non-framework builds, `~/Library/Python/X.Y` for Mac framework builds, and `%APPDATA%\Python` for Windows. This value is used by Distutils to compute the installation directories for scripts, data files, Python modules, etc. for the user installation scheme. See also `PYTHONUSERBASE`.

`site.main()`

Adds all the standard site-specific directories to the module search path. This function is called automatically when this module is imported, unless the Python interpreter was started with the `-S` flag.

Modifié dans la version 3.3 : This function used to be called unconditionally.

`site.addsitedir(sitedir, known_paths=None)`

Add a directory to `sys.path` and process its `.pth` files. Typically used in `sitcustomize` or `usercustomize` (see above).

`site.getsitepackages()`

Return a list containing all global site-packages directories.

Nouveau dans la version 3.2.

`site.getuserbase()`

Return the path of the user base directory, `USER_BASE`. If it is not initialized yet, this function will also set it, respecting `PYTHONUSERBASE`.

Nouveau dans la version 3.2.

`site.getusersitepackages()`

Return the path of the user-specific site-packages directory, `USER_SITE`. If it is not initialized yet, this function will also set it, respecting `PYTHONNOUSERSITE` and `USER_BASE`.

Nouveau dans la version 3.2.

The *site* module also provides a way to get the user directories from the command line :

```
$ python3 -m site --user-site
/home/user/.local/lib/python3.3/site-packages
```

If it is called without arguments, it will print the contents of `sys.path` on the standard output, followed by the value of `USER_BASE` and whether the directory exists, then the same thing for `USER_SITE`, and finally the value of `ENABLE_USER_SITE`.

--user-base

Print the path to the user base directory.

--user-site

Print the path to the user site-packages directory.

If both options are given, user base and user site will be printed (always in this order), separated by `os.pathsep`.

If any option is given, the script will exit with one of these values : 0 if the user site-packages directory is enabled, 1 if it was disabled by the user, 2 if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.

Voir aussi :

PEP 370 – Répertoire site-packages propre à l'utilisateur.

29.14 `fpectl` — Floating point exception control

Note : The `fpectl` module is not built by default, and its usage is discouraged and may be dangerous except in the hands of experts. See also the section *Limitations and other considerations* on limitations for more details.

Most computers carry out floating point operations in conformance with the so-called IEEE-754 standard. On any real computer, some floating point operations produce results that cannot be expressed as a normal floating point value. For example, try

```
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000) / math.exp(1000)
nan
```

(The example above will work on many platforms. DEC Alpha may be one exception.) « Inf » is a special, non-numeric value in IEEE-754 that stands for « infinity », and « nan » means « not a number. » Note that, other than the non-numeric results, nothing special happened when you asked Python to carry out those calculations. That is in fact the default behaviour prescribed in the IEEE-754 standard, and if it works for you, stop reading now.

In some circumstances, it would be better to raise an exception and stop processing at the point where the faulty operation was attempted. The `fpectl` module is for use in that situation. It provides control over floating point units from several hardware manufacturers, allowing the user to turn on the generation of SIGFPE whenever any of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid Operation occurs. In tandem with a pair of wrapper macros that are inserted into the C code comprising your python system, SIGFPE is trapped and converted into the Python `FloatingPointError` exception.

The `fpectl` module defines the following functions and may raise the given exception :

`fpectl.turnon_sigfpe()`

Turn on the generation of SIGFPE, and set up an appropriate signal handler.

`fpectl.turnoff_sigfpe()`

Reset default handling of floating point exceptions.

exception `fpectl.FloatingPointError`

After `turnon_sigfpe()` has been executed, a floating point operation that raises one of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid operation will in turn raise this standard Python exception.

29.14.1 Exemple

The following example demonstrates how to start up and test operation of the `fpectl` module.

```
>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
>>> fpetest.test()
overflow          PASS
FloatingPointError: Overflow

div by 0          PASS
FloatingPointError: Division by zero
[ more output from test elided ]
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: in math_1
```

29.14.2 Limitations and other considerations

Setting up a given processor to trap IEEE-754 floating point errors currently requires custom code on a per-architecture basis. You may have to modify `fpectl` to control your particular hardware.

Conversion of an IEEE-754 exception to a Python exception requires that the wrapper macros `PyFPE_START_PROTECT` and `PyFPE_END_PROTECT` be inserted into your code in an appropriate fashion. Python itself has been modified to support the `fpectl` module, but many other codes of interest to numerical analysts have not.

The `fpectl` module is not thread-safe.

Voir aussi :

Some files in the source distribution may be interesting in learning more about how this module operates. The include file `Include/pyfpe.h` discusses the implementation of this module at some length. `Modules/fpetestmodule.c` gives several examples of use. Many additional examples can be found in `Objects/floatobject.c`.

Interpréteurs Python personnalisés

Les modules décrits dans ce chapitre permettent d'écrire des interfaces similaires à l'interpréteur interactif de Python. Si vous voulez un interpréteur Python qui gère quelques fonctionnalités supplémentaires, vous devriez regarder le module `code`. (Le module `codeop` est un module de plus bas niveau permettant de compiler des morceaux, pas forcément complets, de Python.)

La liste complète des modules décrits dans ce chapitre est :

30.1 `code` — Interpreter base classes

Code source : [Lib/code.py](#)

The `code` module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

class `code.InteractiveInterpreter` (*locals=None*)

This class deals with parsing and interpreter state (the user's namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional *locals* argument specifies the dictionary in which code will be executed; it defaults to a newly created dictionary with key '`__name__`' set to '`__console__`' and key '`__doc__`' set to `None`.

class `code.InteractiveConsole` (*locals=None, filename="<console>"*)

Closely emulate the behavior of the interactive Python interpreter. This class builds on `InteractiveInterpreter` and adds prompting using the familiar `sys.ps1` and `sys.ps2`, and input buffering.

`code.interact` (*banner=None, readfunc=None, local=None, exitmsg=None*)

Convenience function to run a read-eval-print loop. This creates a new instance of `InteractiveConsole` and sets *readfunc* to be used as the `InteractiveConsole.raw_input()` method, if provided. If *local* is provided, it is passed to the `InteractiveConsole` constructor for use as the default namespace for the interpreter loop. The `interact()` method of the instance is then run with *banner* and *exitmsg* passed as the banner and exit message to use, if provided. The console object is discarded after use.

Modifié dans la version 3.6 : Added *exitmsg* parameter.

`code.compile_command(source, filename="<input>", symbol="single")`

This function is useful for programs that want to emulate Python's interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function *almost* always makes the same decision as the real interpreter main loop.

source is the source string; *filename* is the optional filename from which source was read, defaulting to '`<input>`'; and *symbol* is the optional grammar start symbol, which should be either '`single`' (the default) or '`eval`'.

Returns a code object (the same as `compile(source, filename, symbol)`) if the command is complete and valid; `None` if the command is incomplete; raises `SyntaxError` if the command is complete and contains a syntax error, or raises `OverflowError` or `ValueError` if the command contains an invalid literal.

30.1.1 Interactive Interpreter Objects

`InteractiveInterpreter.runsource(source, filename="<input>", symbol="single")`

Compile and run some source in the interpreter. Arguments are the same as for `compile_command()`; the default for *filename* is '`<input>`', and for *symbol* is '`single`'. One several things can happen :

- The input is incorrect; `compile_command()` raised an exception (`SyntaxError` or `OverflowError`). A syntax traceback will be printed by calling the `showsyntaxerror()` method. `runsource()` returns `False`.
- The input is incomplete, and more input is required; `compile_command()` returned `None`. `runsource()` returns `True`.
- The input is complete; `compile_command()` returned a code object. The code is executed by calling the `runcode()` (which also handles run-time exceptions, except for `SystemExit`). `runsource()` returns `False`.

The return value can be used to decide whether to use `sys.ps1` or `sys.ps2` to prompt the next line.

`InteractiveInterpreter.runcode(code)`

Execute a code object. When an exception occurs, `showtraceback()` is called to display a traceback. All exceptions are caught except `SystemExit`, which is allowed to propagate.

A note about `KeyboardInterrupt` : this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

`InteractiveInterpreter.showsyntaxerror(filename=None)`

Display the syntax error that just occurred. This does not display a stack trace because there isn't one for syntax errors. If *filename* is given, it is stuffed into the exception instead of the default filename provided by Python's parser, because it always uses '`<string>`' when reading from a string. The output is written by the `write()` method.

`InteractiveInterpreter.showtraceback()`

Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the `write()` method.

Modifié dans la version 3.5 : The full chained traceback is displayed instead of just the primary traceback.

`InteractiveInterpreter.write(data)`

Write a string to the standard error stream (`sys.stderr`). Derived classes should override this to provide the appropriate output handling as needed.

30.1.2 Interactive Console Objects

The `InteractiveConsole` class is a subclass of `InteractiveInterpreter`, and so offers all the methods of the interpreter objects as well as the following additions.

`InteractiveConsole.interact` (*banner=None, exitmsg=None*)

Closely emulate the interactive Python console. The optional *banner* argument specifies the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter – since it’s so close!).

The optional *exitmsg* argument specifies an exit message printed when exiting. Pass the empty string to suppress the exit message. If *exitmsg* is not given or `None`, a default message is printed.

Modifié dans la version 3.4 : To suppress printing any banner, pass an empty string.

Modifié dans la version 3.6 : Print an exit message when exiting.

`InteractiveConsole.push` (*line*)

Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter’s `runsource()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is `True` if more input is required, `False` if the line was dealt with in some way (this is the same as `runsource()`).

`InteractiveConsole.resetbuffer` ()

Remove any unhandled source text from the input buffer.

`InteractiveConsole.raw_input` (*prompt=""*)

Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, `EOFError` is raised. The base implementation reads from `sys.stdin`; a subclass may replace this with a different implementation.

30.2 codeop — Compilation de code Python

Code source : [Lib/codeop.py](#)

Le module `codeop` fournit des outils permettant d’émuler une boucle de lecture-évaluation-affichage (en anglais *read-eval-print-loop* ou REPL), comme dans le module `code`. Par conséquent, ce module n’est pas destiné à être utilisé directement ; pour inclure un REPL dans un programme, il est préférable d’utiliser le module `code`.

Cette tâche se divise en deux parties :

1. Pouvoir affirmer qu’une ligne d’entrée est une instruction complète, ou achève une instruction : en bref, savoir s’il faut afficher « >>> » ou « . . . » à sa suite.
2. Conserver les instructions déjà entrées par l’utilisateur, afin que les entrées suivantes puissent être compilées avec elles.

Le module `codeop` fournit un moyen d’effectuer ces deux parties, individuellement ou simultanément.

Pour ne faire que la première partie :

`codeop.compile_command` (*source, filename=<input>, symbol="single"*)

Essaye de compiler *source*, qui doit être une chaîne de caractères représentant du code Python valide et renvoie un objet code le cas échéant. Dans ce cas, l’attribut de nom de fichier de l’objet code renvoyé sera *filename* ('<input>' par défaut). Renvoie `None` si *source* n’est pas du code Python valide, mais un *début* de code Python valide.

En cas de problème avec *source*, une exception est levée ; *SyntaxError* si la syntaxe Python est incorrecte, et *OverflowError* ou *ValueError* si un littéral invalide est rencontré.

L'argument *symbol* détermine si *source* est compilée comme une instruction (*'single'*, par défaut) ou comme une *expression* (*'eval'*). Toute autre valeur lèvera *ValueError*.

Note : Il est possible (quoique improbable) que l'analyseur s'arrête avant d'atteindre la fin du code source ; dans ce cas, les symboles venant après peuvent être ignorés au lieu de provoquer une erreur. Par exemple, une barre oblique inverse suivie de deux retours à la ligne peut être suivie par de la mémoire non-initialisée. Ceci sera corrigé quand l'interface de l'analyseur aura été améliorée.

class `codeop.Compile`

Les instances de cette classe ont des méthodes `__call__()` de signature identique à la fonction native `compile()`, à la différence près que si l'instance compile du code source contenant une instruction `__future__`, l'instance s'en « souviendra » et compilera tous les codes sources suivants avec cette instruction activée.

class `codeop.CommandCompiler`

Les instances de cette classe ont des méthodes `__call__()` de signature identique à la fonction `compile_command()`, à la différence près que si l'instance compile du code source contenant une instruction `__future__`, l'instance s'en « souviendra » et compilera tous les codes sources suivants avec cette instruction activée.

Importer des modules

Les modules décrits dans ce chapitre fournissent de nouveaux moyens d'importer d'autres modules Python, et des *hooks* pour personnaliser le processus d'importation.

La liste complète des modules décrits dans ce chapitre est :

31.1 `zipimport` — Importer des modules à partir d'archives Zip

Ce module ajoute la possibilité d'importer des modules Python (`*.py`, `*.pyc`) et des paquets depuis des archives au format ZIP. Il n'est généralement pas nécessaire d'utiliser explicitement le module `zipimport` ; il est automatiquement utilisé par le mécanisme intégré de `import` pour les éléments `sys.path` qui sont des chemins vers les archives ZIP.

Typiquement, `sys.path` est une liste de noms de répertoires sous forme de chaînes. Ce module permet également à un élément de `sys.path` d'être une chaîne nommant une archive de fichier ZIP. L'archive ZIP peut contenir une structure de sous-répertoire pour prendre en charge les importations de paquets, et un chemin dans l'archive peut être spécifié pour importer uniquement à partir d'un sous-répertoire. Par exemple, le chemin d'accès `example.zip/lib/` importerait uniquement depuis le sous-répertoire `lib/` dans l'archive.

Tous les fichiers peuvent être présents dans l'archive ZIP, mais seuls les fichiers `.py` et `.pyc` sont disponibles pour importation. L'importation ZIP des modules dynamiques (`.pyd`, `.so`) est interdite. Notez que si une archive ne contient que des fichiers `.py`, Python n'essaiera pas de modifier l'archive en ajoutant le fichier correspondant `.pyc`, ce qui signifie que si une archive ZIP ne contient pas de fichier `.pyc`, l'importation peut être assez lente.

Les archives ZIP avec un commentaire ne sont actuellement pas prises en charge.

Voir aussi :

PKZIP Application Note Documentation sur le format de fichier ZIP par Phil Katz, créateur du format et des algorithmes utilisés.

PEP 273 - Import Modules from Zip Archives Écrit par James C. Ahlstrom, qui a également fourni une mise en œuvre. Python 2.3 suit les spécifications de PEP 273, mais utilise une implémentation écrite par Just van Rossum qui utilise les crochets d'importation décrits dans PEP 302.

PEP 302 — Nouveaux crochets d'importation Le PEP pour ajouter les crochets d'importation qui aident ce module à fonctionner.

Ce module définit une exception :

exception `zipimport.ZipImportError`

Exception levée par les objets `zipimporter`. C'est une sous-classe de `ImportError`, donc il peut être pris comme `ImportError`, aussi.

31.1.1 Objets `zipimporter`

`zipimporter` est la classe pour importer des fichiers ZIP.

class `zipimport.zipimporter` (*archivepath*)

Créez une nouvelle instance de `zipimporter`. *archivepath* doit être un chemin vers un fichier ZIP, ou vers un chemin spécifique dans un fichier ZIP. Par exemple, un *archivepath* de `foo/bar.zip/lib` cherchera les modules dans le répertoire `lib` du fichier ZIP `foo/bar.zip` (si celui-ci existe).

`ZipImportError` est levée si *archivepath* ne pointe pas vers une archive ZIP valide.

find_module (*fullname* [, *path*])

Rechercher un module spécifié par *fullname*. *fullname* doit être le nom du module entièrement qualifié (*dotted*). Elle retourne l'instance `zipimporter` elle-même si le module a été trouvé, ou `None` si ce n'est pas le cas. L'argument optionnel *path* est ignoré — il est là pour la compatibilité avec le protocole de l'importateur.

get_code (*fullname*)

Retourne l'objet de code pour le module spécifié. Lève `ZipImportError` si le module n'a pas pu être trouvé.

get_data (*pathname*)

Renvoie les données associées à *pathname*. Lève `OSError` si le fichier n'a pas été trouvé.

Modifié dans la version 3.3 : Précédemment, c'était l'exception `IOError` qui était levée, au lieu de `OSError`.

get_filename (*fullname*)

Renvoie la valeur `__file__` qui serait définie si le module spécifié était importé. Lève `ZipImportError` si le module n'a pas pu être trouvé.

Nouveau dans la version 3.1.

get_source (*fullname*)

Renvoie le code source du module spécifié. Lève `ZipImportError` si le module n'a pas pu être trouvé, renvoie `None` si l'archive contient le module, mais n'en a pas la source.

is_package (*fullname*)

Renvoie `True` si le module spécifié par *fullname* est un paquet. Lève `ZipImportError` si le module n'a pas pu être trouvé.

load_module (*fullname*)

Charge le module spécifié par *fullname*. *fullname* doit être le nom du module entièrement qualifié (*dotted*). Il renvoie le module importé, ou augmente `ZipImportError` s'il n'a pas été trouvé.

archive

Le nom de fichier de l'archive ZIP associé à l'importateur, sans sous-chemin possible.

prefix

Le sous-chemin du fichier ZIP où les modules sont recherchés. C'est la chaîne vide pour les objets `zipimporter` qui pointent vers la racine du fichier ZIP.

Les attributs *archive* et *prefix*, lorsqu'ils sont combinés avec une barre oblique, égalent l'argument original *archivepath* donné au constructeur `zipimporter`.

31.1.2 Examples

Voici un exemple qui importe un module d'une archive ZIP — notez que le module `zipimport` n'est pas explicitement utilisé.

```
$ unzip -l example.zip
Archive:  example.zip
  Length      Date    Time    Name
-----
   8467      11-26-02  22:30    jwzthreading.py
-----
   8467                      1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

31.2 pkgutil — Package extension utility

Code source : [Lib/pkgutil.py](#)

This module provides utilities for the import system, in particular package support.

class `pkgutil.ModuleInfo` (*module_finder, name, ispkg*)
 A namedtuple that holds a brief summary of a module's info.
 Nouveau dans la version 3.6.

`pkgutil.extend_path` (*path, name*)
 Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package's `__init__.py`:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

This will add to the package's `__path__` all subdirectories of directories on `sys.path` named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for `*.pkg` files beginning where `*` matches the *name* argument. This feature is similar to `*.pth` files (see the [site](#) module for more information), except that it doesn't special-case lines starting with `import`. A `*.pkg` file is trusted at face value : apart from checking for duplicates, all entries found in a `*.pkg` file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that `sys.path` is a sequence. Items of `sys.path` that are not strings referring to existing directories are ignored. Unicode items on `sys.path` that cause errors when used as filenames may cause this function to raise an exception (in line with `os.path.isdir()` behavior).

class `pkgutil.ImpImporter` (*dirname=None*)
PEP 302 Finder that wraps Python's « classic » import algorithm.
 If *dirname* is a string, a **PEP 302** finder is created that searches that directory. If *dirname* is `None`, a **PEP 302** finder is created that searches the current `sys.path`, plus any modules that are frozen or built-in.

Note that `ImpImporter` does not currently support being used by placement on `sys.meta_path`.

Obsolète depuis la version 3.3 : This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in `importlib`.

class `pkgutil.ImpLoader` (*fullname, file, filename, etc*)

Loader that wraps Python's « classic » import algorithm.

Obsolète depuis la version 3.3 : This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in `importlib`.

`pkgutil.find_loader` (*fullname*)

Retrieve a module *loader* for the given *fullname*.

This is a backwards compatibility wrapper around `importlib.util.find_spec()` that converts most failures to `ImportError` and only returns the loader rather than the full `ModuleSpec`.

Modifié dans la version 3.3 : Updated to be based directly on `importlib` rather than relying on the package internal PEP 302 import emulation.

Modifié dans la version 3.4 : Updated to be based on **PEP 451**

`pkgutil.get_importer` (*path_item*)

Retrieve a *finder* for the given *path_item*.

The returned finder is cached in `sys.path_importer_cache` if it was newly created by a path hook.

The cache (or part of it) can be cleared manually if a rescan of `sys.path_hooks` is necessary.

Modifié dans la version 3.3 : Updated to be based directly on `importlib` rather than relying on the package internal PEP 302 import emulation.

`pkgutil.get_loader` (*module_or_name*)

Get a *loader* object for *module_or_name*.

If the module or package is accessible via the normal import mechanism, a wrapper around the relevant part of that machinery is returned. Returns `None` if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package `__path__`.

Modifié dans la version 3.3 : Updated to be based directly on `importlib` rather than relying on the package internal PEP 302 import emulation.

Modifié dans la version 3.4 : Updated to be based on **PEP 451**

`pkgutil.iter_importers` (*fullname=""*)

Yield *finder* objects for the given module name.

If *fullname* contains a ".", the finders will be for the package containing *fullname*, otherwise they will be all registered top level finders (i.e. those on both `sys.meta_path` and `sys.path_hooks`).

If the named module is in a package, that package is imported as a side effect of invoking this function.

If no module name is specified, all top level finders are produced.

Modifié dans la version 3.3 : Updated to be based directly on `importlib` rather than relying on the package internal PEP 302 import emulation.

`pkgutil.iter_modules` (*path=None, prefix=""*)

Yields `ModuleInfo` for all submodules on *path*, or, if *path* is `None`, all top-level modules on `sys.path`.

path should be either `None` or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note : Only works for a *finder* which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for `importlib.machinery.FileFinder` and `zipimport.zipimporter`.

Modifié dans la version 3.3 : Updated to be based directly on `importlib` rather than relying on the package internal PEP 302 import emulation.

`pkgutil.walk_packages` (*path=None*, *prefix=""*, *onerror=None*)

Yields *ModuleInfo* for all modules recursively on *path*, or, if *path* is *None*, all accessible modules.

path should be either *None* or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note that this function must import all *packages* (not all modules!) on the given *path*, in order to access the `__path__` attribute to find submodules.

onerror is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no *onerror* function is supplied, *ImportErrors* are caught and ignored, while all other exceptions are propagated, terminating the search.

Exemples :

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')

```

Note : Only works for a *finder* which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and *zipimport.zipimporter*.

Modifié dans la version 3.3 : Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

`pkgutil.get_data` (*package*, *resource*)

Get a resource from a package.

This is a wrapper for the *loader* `get_data` API. The *package* argument should be the name of a package, in standard module format (*foo.bar*). The *resource* argument should be in the form of a relative filename, using `/` as the path separator. The parent directory name `..` is not allowed, and nor is a rooted name (starting with a `/`).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of :

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()

```

If the package cannot be located or loaded, or it uses a *loader* which does not support `get_data`, then *None* is returned. In particular, the *loader* for *namespace packages* does not support `get_data`.

31.3 modulefinder — Identifie les modules utilisés par un script

Code source : [Lib/modulefinder.py](#)

Ce module fournit une classe *ModuleFinder* qui peut être utilisée pour déterminer la liste des modules importés par un script. `modulefinder.py` peut aussi être utilisé en tant que script, en passant le nom du fichier Python en argument, ce qui affichera un rapport sur les modules importés.

`modulefinder.AddPackagePath` (*pkg_name*, *path*)

Enregistre que le paquet *pkg_name* peut être trouvé au chemin *path* spécifié.

`modulefinder.ReplacePackage` (*oldname*, *newname*)

Permet de spécifier que le module nommé *oldname* est en réalité le paquet nommé *newname*.

class `modulefinder.ModuleFinder` (*path=None, debug=0, excludes=[], replace_paths=[]*)

Cette classe fournit les méthodes `run_script()` et `report()` pour déterminer l'ensemble des modules importés par un script. *path* peut être une liste de dossiers dans lesquels chercher les modules ; si non spécifié, `sys.path` est utilisé. *debug* définit le niveau de débogage ; des valeurs plus élevées produisent plus de détails sur ce que fait la classe. *excludes* est une liste de noms de modules à exclure de l'analyse. *replace_paths* est une liste de tuples (*oldpath*, *newpath*) qui seront remplacés dans les chemins des modules.

report ()

Affiche un rapport sur la sortie standard qui liste les modules importés par le script et leurs chemins, ainsi que les modules manquants ou qui n'ont pas été trouvés.

run_script (*pathname*)

Analyse le contenu du fichier *pathname*, qui doit contenir du code Python.

modules

Un dictionnaire de correspondance entre nom de modules et modules. Voir *Exemples d'utilisation de la classe ModuleFinder*.

31.3.1 Exemples d'utilisation de la classe `ModuleFinder`

Le script qui sera analysé (*bacon.py*) :

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

Le script qui va afficher le rapport de *bacon.py* :

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(', '.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

Exemple de sortie (peut varier en fonction de l'architecture) :

```
Loaded modules:
_types:
copyreg:  _inverted_registry, _slotnames, __all__
sre_compile:  isstring, _sre, _optimize_unicode
_sre:
sre_constants:  REPEAT_ONE, makedict, AT_END_LINE
```

(suite sur la page suivante)

(suite de la page précédente)

```

sys:
re:  __module__, finditer, _expand
itertools:
__main__:  re, itertools, baconhammeggs
sre_parse:  _PATTERNENDERS, SRE_FLAG_UNICODE
array:
types:  __module__, IntType, TypeType
-----
Modules not imported:
guido.python.ham
baconhammeggs

```

31.4 runpy — Locating and executing Python modules

Code source : [Lib/runpy.py](#)

The *runpy* module is used to locate and run Python modules without importing them first. Its main use is to implement the `-m` command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

Note that this is *not* a sandbox module - all code is executed in the current process, and any side effects (such as cached imports of other modules) will remain in place after the functions have returned.

Furthermore, any functions and classes defined by the executed code are not guaranteed to work correctly after a *runpy* function has returned. If that limitation is not acceptable for a given use case, *importlib* is likely to be a more suitable choice than this module.

The *runpy* module provides two functions :

`runpy.run_module(mod_name, init_globals=None, run_name=None, alter_sys=False)`

Execute the code of the specified module and return the resulting module globals dictionary. The module's code is first located using the standard import mechanism (refer to [PEP 302](#) for details) and then executed in a fresh module namespace.

The `mod_name` argument should be an absolute module name. If the module name refers to a package rather than a normal module, then that package is imported and the `__main__` submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument `init_globals` may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by `run_module()`.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

`__name__` is set to `run_name` if this optional argument is not *None*, to `mod_name + '.__main__'` if the named module is a package and to the `mod_name` argument otherwise.

`__spec__` will be set appropriately for the *actually* imported module (that is, `__spec__.name` will always be `mod_name` or `mod_name + '.__main__'`, never `run_name`).

`__file__`, `__cached__`, `__loader__` and `__package__` are set as normal based on the module spec.

If the argument `alter_sys` is supplied and evaluates to *True*, then `sys.argv[0]` is updated with the value of `__file__` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. Both `sys.argv[0]` and `sys.modules[__name__]` are restored to their original values before the function returns.

Note that this manipulation of `sys` is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the `sys` module be left alone when invoking this function from threaded code.

Voir aussi :

The `-m` option offering equivalent functionality from the command line.

Modifié dans la version 3.1 : Added ability to execute packages by looking for a `__main__` submodule.

Modifié dans la version 3.2 : Added `__cached__` global variable (see [PEP 3147](#)).

Modifié dans la version 3.4 : Updated to take advantage of the module spec feature added by [PEP 451](#). This allows `__cached__` to be set correctly for modules run this way, as well as ensuring the real module name is always accessible as `__spec__.name`.

`runpy.run_path(file_path, init_globals=None, run_name=None)`

Execute the code at the named filesystem location and return the resulting module globals dictionary. As with a script name supplied to the CPython command line, the supplied path may refer to a Python source file, a compiled bytecode file or a valid `sys.path` entry containing a `__main__` module (e.g. a zipfile containing a top-level `__main__.py` file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid `sys.path` entry (typically a zipfile or directory), the entry is first added to the beginning of `sys.path`. The function then looks for and executes a `__main__` module using the updated path. Note that there is no special protection against invoking an existing `__main__` entry located elsewhere on `sys.path` if there is no such module at the specified location. The optional dictionary argument `init_globals` may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by `run_path()`.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

`__name__` is set to `run_name` if this optional argument is not `None` and to `'<run_path>'` otherwise.

If the supplied path directly references a script file (whether as source or as precompiled byte code), then `__file__` will be set to the supplied path, and `__spec__`, `__cached__`, `__loader__` and `__package__` will all be set to `None`.

If the supplied path is a reference to a valid `sys.path` entry, then `__spec__` will be set appropriately for the imported `__main__` module (that is, `__spec__.name` will always be `__main__`). `__file__`, `__cached__`, `__loader__` and `__package__` will be set as normal based on the module spec.

A number of alterations are also made to the `sys` module. Firstly, `sys.path` may be altered as described above. `sys.argv[0]` is updated with the value of `file_path` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. All modifications to items in `sys` are reverted before the function returns.

Note that, unlike `run_module()`, the alterations made to `sys` are not optional in this function as these adjustments are essential to allowing the execution of `sys.path` entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

Voir aussi :

using-on-interface-options for equivalent functionality on the command line (`python path/to/script`).

Nouveau dans la version 3.2.

Modifié dans la version 3.4 : Updated to take advantage of the module spec feature added by [PEP 451](#). This allows `__cached__` to be set correctly in the case where `__main__` is imported from a valid `sys.path` entry rather than being executed directly.

Voir aussi :

PEP 338 – Exécuter des modules en tant que scripts PEP written and implemented by Nick Coghlan.

PEP 366 – Main module explicit relative imports PEP written and implemented by Nick Coghlan.

PEP 451 – A ModuleSpec Type for the Import System PEP written and implemented by Eric Snow

using-on-general - CPython command line details

The `importlib.import_module()` function

31.5 importlib — The implementation of import

Nouveau dans la version 3.1.

Source code : [Lib/importlib/__init__.py](#)

31.5.1 Introduction

The purpose of the `importlib` package is two-fold. One is to provide the implementation of the `import` statement (and thus, by extension, the `__import__()` function) in Python source code. This provides an implementation of `import` which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement `import` are exposed in this package, making it easier for users to create their own custom objects (known generically as an *importer*) to participate in the import process.

Voir aussi :

import The language reference for the `import` statement.

Packages specification Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on `None` in `sys.modules`).

La fonction `__import__()` The `import` statement is syntactic sugar for this function.

PEP 235 Import on Case-Insensitive Platforms

PEP 263 Defining Python Source Code Encodings

PEP 302 New Import Hooks

PEP 328 Imports : Multi-Line and Absolute/Relative

PEP 366 Main module explicit relative imports

PEP 420 Implicit namespace packages

PEP 451 A ModuleSpec Type for the Import System

PEP 488 Elimination of PYO files

PEP 489 Multi-phase extension module initialization

PEP 3120 Using UTF-8 as the Default Source Encoding

PEP 3147 PYC Repository Directories

31.5.2 Fonctions

`importlib.__import__(name, globals=None, locals=None, fromlist=(), level=0)`

An implementation of the built-in `__import__()` function.

Note : Programmatic importing of modules should use `import_module()` instead of this function.

`importlib.import_module(name, package=None)`

Import a module. The *name* argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`). If the name is specified in relative terms, then the *package* argument must be set to the name of the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`).

The `import_module()` function acts as a simplifying wrapper around `importlib.__import__()`. This means all semantics of the function are derived from `importlib.__import__()`. The most important difference between these two functions is that `import_module()` returns the specified package or module (e.g. `pkg.mod`), while `__import__()` returns the top-level package or module (e.g. `pkg`).

If you are dynamically importing a module that was created since the interpreter began execution (e.g., created a Python source file), you may need to call `invalidate_caches()` in order for the new module to be noticed by the import system.

Modifié dans la version 3.3 : Parent packages are automatically imported.

`importlib.find_loader(name, path=None)`

Find the loader for a module, optionally within the specified *path*. If the module is in `sys.modules`, then `sys.modules[name].__loader__` is returned (unless the loader would be `None` or is not set, in which case `ValueError` is raised). Otherwise a search using `sys.meta_path` is done. `None` is returned if no loader is found.

A dotted name does not have its parents implicitly imported as that requires loading them and that may not be desired. To properly import a submodule you will need to import all parent packages of the submodule and use the correct argument to *path*.

Nouveau dans la version 3.3.

Modifié dans la version 3.4 : If `__loader__` is not set, raise `ValueError`, just like when the attribute is set to `None`.

Obsolète depuis la version 3.4 : Use `importlib.util.find_spec()` instead.

`importlib.invalidate_caches()`

Invalidate the internal caches of finders stored at `sys.meta_path`. If a finder implements `invalidate_caches()` then it will be called to perform the invalidation. This function should be called if any modules are created/installed while your program is running to guarantee all finders will notice the new module's existence.

Nouveau dans la version 3.3.

`importlib.reload(module)`

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (which can be different if re-importing causes a different object to be placed in `sys.modules`).

When `reload()` is executed :

- Python module's code is recompiled and the module-level code re-executed, defining a new set of objects which are bound to names in the module's dictionary by reusing the *loader* which originally loaded the module. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats :

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a `try` statement it can test for the table's presence and skip its initialization if desired :

```
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading `sys`, `__main__`, `builtins` and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from ... import ...`, calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (`module.name`) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

Nouveau dans la version 3.4.

31.5.3 `importlib.abc` – Abstract base classes related to import

Source code : [Lib/importlib/abc.py](#)

The `importlib.abc` module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC hierarchy :

```
object
+-- Finder (deprecated)
|   +-- MetaPathFinder
|   +-- PathEntryFinder
+-- Loader
    +-- ResourceLoader -----+
    +-- InspectLoader         |
        +-- ExecutionLoader --+
                                +-- FileLoader
                                +-- SourceLoader
```

class `importlib.abc.Finder`

An abstract base class representing a *finder*.

Obsolète depuis la version 3.3 : Use `MetaPathFinder` or `PathEntryFinder` instead.

abstractmethod `find_module(fullname, path=None)`

An abstract method for finding a *loader* for the specified module. Originally specified in [PEP 302](#), this method was meant for use in `sys.meta_path` and in the path-based import subsystem.

Modifié dans la version 3.4 : Returns `None` when called instead of raising `NotImplementedError`.

class `importlib.abc.MetaPathFinder`

An abstract base class representing a *meta path finder*. For compatibility, this is a subclass of `Finder`.

Nouveau dans la version 3.3.

find_spec (`fullname, path, target=None`)

An abstract method for finding a *spec* for the specified module. If this is a top-level import, `path` will be `None`. Otherwise, this is a search for a subpackage or module and `path` will be the value of `__path__` from the parent package. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return.

Nouveau dans la version 3.4.

find_module (*fullname*, *path*)

A legacy method for finding a *loader* for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of `__path__` from the parent package. If a loader cannot be found, `None` is returned.

If *find_spec()* is defined, backwards-compatible functionality is provided.

Modifié dans la version 3.4 : Returns `None` when called instead of raising `NotImplementedError`. Can use *find_spec()* to provide functionality.

Obsolète depuis la version 3.4 : Use *find_spec()* instead.

invalidate_caches ()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by `importlib.invalidate_caches()` when invalidating the caches of all finders on `sys.meta_path`.

Modifié dans la version 3.4 : Returns `None` when called instead of `NotImplemented`.

class `importlib.abc.PathEntryFinder`

An abstract base class representing a *path entry finder*. Though it bears some similarities to *MetaPathFinder*, `PathEntryFinder` is meant for use only within the path-based import subsystem provided by `PathFinder`. This ABC is a subclass of *Finder* for compatibility reasons only.

Nouveau dans la version 3.3.

find_spec (*fullname*, *target=None*)

An abstract method for finding a *spec* for the specified module. The finder will search for the module only within the *path entry* to which it is assigned. If a spec cannot be found, `None` is returned. When passed in, *target* is a module object that the finder may use to make a more educated guess about what spec to return.

Nouveau dans la version 3.4.

find_loader (*fullname*)

A legacy method for finding a *loader* for the specified module. Returns a 2-tuple of (*loader*, *portion*) where *portion* is a sequence of file system locations contributing to part of a namespace package. The loader may be `None` while specifying *portion* to signify the contribution of the file system locations to a namespace package. An empty list can be used for *portion* to signify the loader is not part of a namespace package. If *loader* is `None` and *portion* is the empty list then no loader or location for a namespace package were found (i.e. failure to find anything for the module).

If *find_spec()* is defined then backwards-compatible functionality is provided.

Modifié dans la version 3.4 : Returns (`None`, `[]`) instead of raising `NotImplementedError`. Uses *find_spec()* when available to provide functionality.

Obsolète depuis la version 3.4 : Use *find_spec()* instead.

find_module (*fullname*)

A concrete implementation of *Finder.find_module()* which is equivalent to `self.find_loader(fullname)[0]`.

Obsolète depuis la version 3.4 : Use *find_spec()* instead.

invalidate_caches ()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by `PathFinder.invalidate_caches()` when invalidating the caches of all cached finders.

class `importlib.abc.Loader`

An abstract base class for a *loader*. See [PEP 302](#) for the exact definition for a loader.

create_module (*spec*)

A method that returns the module object to use when importing a module. This method may return `None`, indicating that default module creation semantics should take place.

Nouveau dans la version 3.4.

Modifié dans la version 3.5 : Starting in Python 3.6, this method will not be optional when *exec_module()* is defined.

exec_module (*module*)

An abstract method that executes the module in its own namespace when a module is imported or reloaded. The module should already be initialized when `exec_module()` is called. When this method exists, `create_module()` must be defined.

Nouveau dans la version 3.4.

Modifié dans la version 3.6 : `create_module()` must also be defined.

load_module (*fullname*)

A legacy method for loading a module. If the module cannot be loaded, `ImportError` is raised, otherwise the loaded module is returned.

If the requested module already exists in `sys.modules`, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into `sys.modules` before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from `sys.modules`; modules already in `sys.modules` before the loader began execution should be left alone (see `importlib.util.module_for_loader()`).

The loader should set several attributes on the module. (Note that some of these attributes can change when a module is reloaded) :

- **__name__** The name of the module.
- **__file__** The path to where the module data is stored (not set for built-in modules).
- **__cached__** The path to where a compiled version of the module is/should be stored (not set when the attribute would be inappropriate).
- **__path__** A list of strings specifying the search path within a package. This attribute is not set on modules.
- **__package__** The parent package for the module/package. If the module is top-level then it has a value of the empty string. The `importlib.util.module_for_loader()` decorator can handle the details for `__package__`.
- **__loader__** The loader used to load the module. The `importlib.util.module_for_loader()` decorator can handle the details for `__package__`.

When `exec_module()` is available then backwards-compatible functionality is provided.

Modifié dans la version 3.4 : Raise `ImportError` when called instead of `NotImplementedError`. Functionality provided when `exec_module()` is available.

Obsolète depuis la version 3.4 : The recommended API for loading a module is `exec_module()` (and `create_module()`). Loaders should implement it instead of `load_module()`. The import machinery takes care of all the other responsibilities of `load_module()` when `exec_module()` is implemented.

module_repr (*module*)

A legacy method which when implemented calculates and returns the given module's repr, as a string. The module type's default `repr()` will use the result of this method as appropriate.

Nouveau dans la version 3.3.

Modifié dans la version 3.4 : Made optional instead of an abstractmethod.

Obsolète depuis la version 3.4 : The import machinery now takes care of this automatically.

class `importlib.abc.ResourceLoader`

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loading arbitrary resources from the storage back-end.

abstractmethod `get_data` (*path*)

An abstract method to return the bytes for the data located at *path*. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. `OSError` is to be raised if the *path* cannot be found. The *path* is expected to be constructed using a module's `__file__` attribute or an item from a package's `__path__`.

Modifié dans la version 3.4 : Raises `OSError` instead of `NotImplementedError`.

class `importlib.abc.InspectLoader`

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loaders that inspect modules.

get_code (*fullname*)

Return the code object for a module, or `None` if the module does not have a code object (as would be the case, for example, for a built-in module). Raise an `ImportError` if loader cannot find the requested module.

Note : While the method has a default implementation, it is suggested that it be overridden if possible for performance.

Modifié dans la version 3.4 : No longer abstract and a concrete implementation is provided.

abstractmethod get_source (*fullname*)

An abstract method to return the source of a module. It is returned as a text string using *universal newlines*, translating all recognized line separators into `'\n'` characters. Returns `None` if no source is available (e.g. a built-in module). Raises `ImportError` if the loader cannot find the module specified.

Modifié dans la version 3.4 : Raises `ImportError` instead of `NotImplementedError`.

is_package (*fullname*)

An abstract method to return a true value if the module is a package, a false value otherwise. `ImportError` is raised if the *loader* cannot find the module.

Modifié dans la version 3.4 : Raises `ImportError` instead of `NotImplementedError`.

static source_to_code (*data*, *path*=*'<string>'*)

Create a code object from Python source.

The *data* argument can be whatever the `compile()` function supports (i.e. string or bytes). The *path* argument should be the « path » to where the source code originated from, which can be an abstract concept (e.g. location in a zip file).

With the subsequent code object one can execute it in a module by running `exec(code, module.__dict__)`.

Nouveau dans la version 3.4.

Modifié dans la version 3.5 : Made the method static.

exec_module (*module*)

Implementation of `Loader.exec_module()`.

Nouveau dans la version 3.4.

load_module (*fullname*)

Implementation of `Loader.load_module()`.

Obsolète depuis la version 3.4 : use `exec_module()` instead.

class `importlib.abc.ExecutionLoader`

An abstract base class which inherits from `InspectLoader` that, when implemented, helps a module to be executed as a script. The ABC represents an optional **PEP 302** protocol.

abstractmethod get_filename (*fullname*)

An abstract method that is to return the value of `__file__` for the specified module. If no path is available, `ImportError` is raised.

If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

Modifié dans la version 3.4 : Raises `ImportError` instead of `NotImplementedError`.

class `importlib.abc.FileLoader` (*fullname*, *path*)

An abstract base class which inherits from `ResourceLoader` and `ExecutionLoader`, providing concrete implementations of `ResourceLoader.get_data()` and `ExecutionLoader.get_filename()`.

The *fullname* argument is a fully resolved name of the module the loader is to handle. The *path* argument is the path to the file for the module.

Nouveau dans la version 3.3.

name

The name of the module the loader can handle.

path

Path to the file of the module.

load_module (*fullname*)

Calls super's `load_module()`.

Obsolète depuis la version 3.4 : Use `Loader.exec_module()` instead.

abstractmethod `get_filename` (*fullname*)

Returns *path*.

abstractmethod `get_data` (*path*)

Reads *path* as a binary file and returns the bytes from it.

class `importlib.abc.SourceLoader`

An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both `ResourceLoader` and `ExecutionLoader`, requiring the implementation of :

— `ResourceLoader.get_data()`

— `ExecutionLoader.get_filename()` Should only return the path to the source file ; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods (or causing them to raise `NotImplementedError`) causes the loader to only work with source code. Implementing the methods allows the loader to work with source *and* bytecode files ; it does not allow for *sourceless* loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler, and so no bytecode-specific API is exposed.

path_stats (*path*)

Optional abstract method which returns a *dict* containing metadata about the specified path. Supported dictionary keys are :

— 'mtime' (mandatory) : an integer or floating-point number representing the modification time of the source code ;

— 'size' (optional) : the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. If the path cannot be handled, `OSError` is raised.

Nouveau dans la version 3.3.

Modifié dans la version 3.4 : Raise `OSError` instead of `NotImplementedError`.

path_mtime (*path*)

Optional abstract method which returns the modification time for the specified path.

Obsolète depuis la version 3.3 : This method is deprecated in favour of `path_stats()`. You don't have to implement it, but it is still available for compatibility purposes. Raise `OSError` if the path cannot be handled.

Modifié dans la version 3.4 : Raise `OSError` instead of `NotImplementedError`.

set_data (*path*, *data*)

Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (`errno.EACCES/PermissionError`), do not propagate the exception.

Modifié dans la version 3.4 : No longer raises `NotImplementedError` when called.

get_code (*fullname*)

Concrete implementation of `InspectLoader.get_code()`.

exec_module (*module*)

Concrete implementation of `Loader.exec_module()`.

Nouveau dans la version 3.4.

load_module (*fullname*)

Concrete implementation of `Loader.load_module()`.

Obsolète depuis la version 3.4 : Use `exec_module()` instead.

get_source (*fullname*)

Concrete implementation of `InspectLoader.get_source()`.

is_package (*fullname*)

Concrete implementation of *InspectLoader.is_package()*. A module is determined to be a package if its file path (as provided by *ExecutionLoader.get_filename()*) is a file named `__init__` when the file extension is removed **and** the module name itself does not end in `__init__`.

31.5.4 `importlib.machinery` – Importers and path hooks

Source code : [Lib/importlib/machinery.py](#)

This module contains the various objects that help `import` find and load modules.

`importlib.machinery.SOURCE_SUFFIXES`

A list of strings representing the recognized file suffixes for source modules.

Nouveau dans la version 3.3.

`importlib.machinery.DEBUG_BYTECODE_SUFFIXES`

A list of strings representing the file suffixes for non-optimized bytecode modules.

Nouveau dans la version 3.3.

Obsolète depuis la version 3.5 : Use *BYTECODE_SUFFIXES* instead.

`importlib.machinery.OPTIMIZED_BYTECODE_SUFFIXES`

A list of strings representing the file suffixes for optimized bytecode modules.

Nouveau dans la version 3.3.

Obsolète depuis la version 3.5 : Use *BYTECODE_SUFFIXES* instead.

`importlib.machinery.BYTECODE_SUFFIXES`

A list of strings representing the recognized file suffixes for bytecode modules (including the leading dot).

Nouveau dans la version 3.3.

Modifié dans la version 3.5 : The value is no longer dependent on `__debug__`.

`importlib.machinery.EXTENSION_SUFFIXES`

A list of strings representing the recognized file suffixes for extension modules.

Nouveau dans la version 3.3.

`importlib.machinery.all_suffixes()`

Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, *inspect.getmodulename()*).

Nouveau dans la version 3.3.

class `importlib.machinery.BuiltinImporter`

An *importer* for built-in modules. All known built-in modules are listed in *sys.builtin_module_names*. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Modifié dans la version 3.5 : As part of **PEP 489**, the builtin importer now implements *Loader.create_module()* and *Loader.exec_module()*

class `importlib.machinery.FrozenImporter`

An *importer* for frozen modules. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

class `importlib.machinery.WindowsRegistryFinder`

Finder for modules declared in the Windows registry. This class implements the `importlib.abc.MetaPathFinder` ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

Nouveau dans la version 3.3.

Obsolète depuis la version 3.6 : Use `site` configuration instead. Future versions of Python may not enable this finder by default.

class `importlib.machinery.PathFinder`

A *Finder* for `sys.path` and package `__path__` attributes. This class implements the `importlib.abc.MetaPathFinder` ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod `find_spec` (*fullname*, *path=None*, *target=None*)

Class method that attempts to find a *spec* for the module specified by *fullname* on `sys.path` or, if defined, on *path*. For each path entry that is searched, `sys.path_importer_cache` is checked. If a non-false object is found then it is used as the *path entry finder* to look for the module being searched for. If no entry is found in `sys.path_importer_cache`, then `sys.path_hooks` is searched for a finder for the path entry and, if found, is stored in `sys.path_importer_cache` along with being queried about the module. If no finder is ever found then `None` is both stored in the cache and returned.

Nouveau dans la version 3.4.

Modifié dans la version 3.5 : If the current working directory – represented by an empty string – is no longer valid then `None` is returned but no value is cached in `sys.path_importer_cache`.

classmethod `find_module` (*fullname*, *path=None*)

A legacy wrapper around `find_spec()`.

Obsolète depuis la version 3.4 : Use `find_spec()` instead.

classmethod `invalidate_caches` ()

Calls `importlib.abc.PathEntryFinder.invalidate_caches()` on all finders stored in `sys.path_importer_cache`.

Modifié dans la version 3.4 : Calls objects in `sys.path_hooks` with the current working directory for `' '` (i.e. the empty string).

class `importlib.machinery.FileFinder` (*path*, **loader_details*)

A concrete implementation of `importlib.abc.PathEntryFinder` which caches results from the file system.

The *path* argument is the directory for which the finder is in charge of searching.

The *loader_details* argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call `importlib.invalidate_caches()`.

Nouveau dans la version 3.3.

path

The path the finder will search in.

find_spec (*fullname*, *target=None*)

Attempt to find the spec to handle *fullname* within *path*.

Nouveau dans la version 3.4.

find_loader (*fullname*)

Attempt to find the loader to handle *fullname* within *path*.

invalidate_caches()

Clear out the internal cache.

classmethod path_hook(*loader_details)

A class method which returns a closure for use on `sys.path_hooks`. An instance of `FileFinder` is returned by the closure using the path argument given to the closure directly and `loader_details` indirectly.

If the argument to the closure is not an existing directory, `ImportError` is raised.

class `importlib.machinery.SourceFileLoader` (*fullname, path*)

A concrete implementation of `importlib.abc.SourceLoader` by subclassing `importlib.abc.FileLoader` and providing some concrete implementations of other methods.

Nouveau dans la version 3.3.

name

The name of the module that this loader will handle.

path

The path to the source file.

is_package (*fullname*)

Return true if *path* appears to be for a package.

path_stats (*path*)

Concrete implementation of `importlib.abc.SourceLoader.path_stats()`.

set_data (*path, data*)

Concrete implementation of `importlib.abc.SourceLoader.set_data()`.

load_module (*name=None*)

Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

Obsolète depuis la version 3.6 : Use `importlib.abc.Loader.exec_module()` instead.

class `importlib.machinery.SourcelessFileLoader` (*fullname, path*)

A concrete implementation of `importlib.abc.FileLoader` which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

Nouveau dans la version 3.3.

name

The name of the module the loader will handle.

path

The path to the bytecode file.

is_package (*fullname*)

Determines if the module is a package based on *path*.

get_code (*fullname*)

Returns the code object for *name* created from *path*.

get_source (*fullname*)

Returns None as bytecode files have no source when this loader is used.

load_module (*name=None*)

Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

Obsolète depuis la version 3.6 : Use `importlib.abc.Loader.exec_module()` instead.

class `importlib.machinery.ExtensionFileLoader` (*fullname, path*)

A concrete implementation of `importlib.abc.ExecutionLoader` for extension modules.

The *fullname* argument specifies the name of the module the loader is to support. The *path* argument is the path to the extension module's file.

Nouveau dans la version 3.3.

has_location

Boolean indicating whether or not the module's « origin » attribute refers to a loadable location.

31.5.5 `importlib.util` – Utility code for importers

Source code : [Lib/importlib/util.py](#)

This module contains the various objects that help in the construction of an *importer*.

`importlib.util.MAGIC_NUMBER`

The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider `importlib.abc.SourceLoader`.

Nouveau dans la version 3.4.

`importlib.util.cache_from_source` (*path*, *debug_override*=None, *, *optimization*=None)

Return the [PEP 3147/PEP 488](#) path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see `get_tag()` ; if `sys.implementation.cache_tag` is not defined then `NotImplementedError` will be raised).

The *optimization* parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so `/foo/bar/baz.py` with an *optimization* of `' '` will result in a bytecode path of `/foo/bar/__pycache__/baz.cpython-32.pyc`. None causes the interpreter's optimization level to be used. Any other value's string representation being used, so `/foo/bar/baz.py` with an *optimization* of `2` will lead to the bytecode path of `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`. The string representation of *optimization* can only be alphanumeric, else `ValueError` is raised.

The *debug_override* parameter is deprecated and can be used to override the system's value for `__debug__`. A `True` value is the equivalent of setting *optimization* to the empty string. A `False` value is the same as setting *optimization* to `1`. If both *debug_override* and *optimization* are not `None` then `TypeError` is raised.

Nouveau dans la version 3.4.

Modifié dans la version 3.5 : The *optimization* parameter was added and the *debug_override* parameter was deprecated.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`importlib.util.source_from_cache` (*path*)

Given the *path* to a [PEP 3147](#) file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to [PEP 3147](#) or [PEP 488](#) format, a `ValueError` is raised. If `sys.implementation.cache_tag` is not defined, `NotImplementedError` is raised.

Nouveau dans la version 3.4.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`importlib.util.decode_source` (*source_bytes*)

Decode the given bytes representing source code and return it as a string with universal newlines (as required by `importlib.abc.InspectLoader.get_source()`).

Nouveau dans la version 3.4.

`importlib.util.resolve_name` (*name*, *package*)

Resolve a relative module name to an absolute one.

If *name* has no leading dots, then *name* is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __package__)` without doing a check to see if the *package* argument is needed. `ValueError` is raised if *name* is a relative module name but *package* is a false value (e.g. `None` or the empty string). `ValueError` is also raised a relative name would escape its containing package (e.g. requesting `..bacon` from within the `spam` package).

Nouveau dans la version 3.3.

`importlib.util.find_spec(name, package=None)`

Find the *spec* for a module, optionally relative to the specified **package** name. If the module is in `sys.modules`, then `sys.modules[name].__spec__` is returned (unless the spec would be `None` or is not set, in which case `ValueError` is raised). Otherwise a search using `sys.meta_path` is done. `None` is returned if no spec is found.

If **name** is for a submodule (contains a dot), the parent module is automatically imported.

name and **package** work the same as for `import_module()`.

Nouveau dans la version 3.4.

`importlib.util.module_from_spec(spec)`

Create a new module based on **spec** and `spec.loader.create_module()`.

If `spec.loader.create_module` does not return `None`, then any pre-existing attributes will not be reset. Also, no `AttributeError` will be raised if triggered while accessing **spec** or setting an attribute on the module.

This function is preferred over using `types.ModuleType` to create a new module as **spec** is used to set as many import-controlled attributes on the module as possible.

Nouveau dans la version 3.5.

`@importlib.util.module_for_loader`

A *decorator* for `importlib.abc.Loader.load_module()` to handle selecting the proper module object to load with. The decorated method is expected to have a call signature taking two positional arguments (e.g. `load_module(self, module)`) for which the second argument will be the module **object** to be used by the loader. Note that the decorator will not work on static methods because of the assumption of two arguments.

The decorated method will take in the **name** of the module to be loaded as expected for a *loader*. If the module is not found in `sys.modules` then a new one is constructed. Regardless of where the module came from, `__loader__` set to **self** and `__package__` is set based on what `importlib.abc.InspectLoader.is_package()` returns (if available). These attributes are set unconditionally to support reloading.

If an exception is raised by the decorated method and a module was added to `sys.modules`, then the module will be removed to prevent a partially initialized module from being left in `sys.modules`. If the module was already in `sys.modules` then it is left alone.

Modifié dans la version 3.3 : `__loader__` and `__package__` are automatically set (when possible).

Modifié dans la version 3.4 : Set `__name__`, `__loader__` `__package__` unconditionally to support reloading.

Obsolète depuis la version 3.4 : The import machinery now directly performs all the functionality provided by this function.

`@importlib.util.set_loader`

A *decorator* for `importlib.abc.Loader.load_module()` to set the `__loader__` attribute on the returned module. If the attribute is already set the decorator does nothing. It is assumed that the first positional argument to the wrapped method (i.e. `self`) is what `__loader__` should be set to.

Modifié dans la version 3.4 : Set `__loader__` if set to `None`, as if the attribute does not exist.

Obsolète depuis la version 3.4 : The import machinery takes care of this automatically.

`@importlib.util.set_package`

A *decorator* for `importlib.abc.Loader.load_module()` to set the `__package__` attribute on the returned module. If `__package__` is set and has a value other than `None` it will not be changed.

Obsolète depuis la version 3.4 : The import machinery takes care of this automatically.

`importlib.util.spec_from_loader(name, loader, *, origin=None, is_package=None)`

A factory function for creating a `ModuleSpec` instance based on a loader. The parameters have the same meaning as they do for `ModuleSpec`. The function uses available *loader* APIs, such as `InspectLoader.is_package()`, to fill in any missing information on the spec.

Nouveau dans la version 3.4.

`importlib.util.spec_from_file_location(name, location, *, loader=None, submodule_search_locations=None)`

A factory function for creating a `ModuleSpec` instance based on the path to a file. Missing information will be filled in on the spec by making use of loader APIs and by the implication that the module will be file-based.

Nouveau dans la version 3.4.

Modifié dans la version 3.6 : Accepte un *path-like object*.

class `importlib.util.LazyLoader` (*loader*)

A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define `exec_module()` as control over what module type is used for the module is required. For those same reasons, the loader's `create_module()` method must return `None` or a type for which its `__class__` attribute can be mutated along with not using *slots*. Finally, modules which substitute the object placed into `sys.modules` will not work as there is no way to properly replace the module references throughout the interpreter safely; `ValueError` is raised if such a substitution is detected.

Note : For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

Nouveau dans la version 3.5.

Modifié dans la version 3.6 : Began calling `create_module()`, removing the compatibility warning for `importlib.machinery.BuiltinImporter` and `importlib.machinery.ExtensionFileLoader`.

classmethod `factory` (*loader*)

A static method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

31.5.6 Exemples

Importing programmatically

To programmatically import a module, use `importlib.import_module()`.

```
import importlib

itertools = importlib.import_module('itertools')
```

Checking if a module can be imported

If you need to find out if a module can be imported without actually doing the import, then you should use `importlib.util.find_spec()`.

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

spec = importlib.util.find_spec(name)
```

(suite sur la page suivante)

(suite de la page précédente)

```

if spec is None:
    print("can't find the itertools module")
else:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    # Adding the module to sys.modules is optional.
    sys.modules[name] = module

```

Importing a source file directly

To import a Python source file directly, use the following recipe (Python 3.5 and newer only) :

```

import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name, file_path)
module = importlib.util.module_from_spec(spec)
spec.loader.exec_module(module)
# Optional; only necessary if you want to be able to import the module
# by name later.
sys.modules[module_name] = module

```

Setting up an importer

For deep customizations of import, you typically want to implement an *importer*. This means managing both the *finder* and *loader* side of things. For finders there are two flavours to choose from depending on your needs : a *meta path finder* or a *path entry finder*. The former is what you would put on `sys.meta_path` while the latter is what you create using a *path entry hook* on `sys.path_hooks` which works with `sys.path` entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package) :

```

import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms

```

(suite sur la page suivante)

(suite de la page précédente)

```
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through `importlib`. The following helps illustrate the various APIs that `importlib` exposes by providing an approximate implementation of `importlib.import_module()` (Python 3.4 and newer for the `importlib` usage, Python 3.6 and newer for other parts of the code).

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    sys.modules[absolute_name] = module
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

Services du Langage Python

Python fournit quelques modules pour vous aider à travailler avec le langage Python lui-même. Ces modules gèrent entre autres l'analyse lexicale, l'analyse syntaxique, et le désassemblage de *bytecode*.

Ces modules sont :

32.1 `parser` — Accès aux arbres syntaxiques

Le module `parser` expose une interface à l'analyseur et au compilateur de byte-code internes de Python. Son objectif principal est de permettre à du code Python de modifier l'arbre syntaxique d'une expression Python puis de la rendre exécutable. Cette approche est plus fiable que celle consistant à manipuler des chaînes de caractères, puisque l'analyse est faite avec le même analyseur que celui utilisé pour le code de l'application. C'est aussi plus rapide.

Note : À partir de Python 2.5, il est plus pratique de faire ces manipulations entre la génération de l'AST (*Abstract Syntax Tree*) et la compilation, en utilisant le module `ast`.

Certaines particularités de ce module sont importantes à retenir pour en faire un bon usage. Ce n'est pas un tutoriel sur la modification d'arbres syntaxiques Python, mais certains exemples d'utilisation du module `parser` sont présentés.

Le prérequis le plus important est une bonne compréhension de la grammaire Python utilisée par l'analyseur interne dont la syntaxe est documentée exhaustivement dans `reference-index`. L'analyseur lui-même est généré à partir d'une grammaire spécifiée dans le fichier `Grammar/Grammar` dans la distribution standard de Python. Les arbres syntaxiques stockés dans les objets ST créés par les fonctions `expr()` ou `suite()` de ce module sont directement le résultat de l'analyseur interne, alors que les objets ST créés par `sequence2st()` simulent ces structures. N'oubliez pas qu'une séquence considérée « correcte » dans une version de Python peut ne pas l'être dans une autre, la grammaire de Python pouvant évoluer. Cependant, déplacer du code source d'une version de Python à une autre sous forme textuelle donnera toujours des arbres syntaxique corrects, à l'exception qu'une version plus ancienne de l'interpréteur ne pourra pas analyser les constructions récentes du langage. Les arbres syntaxiques quant à eux ne sont généralement pas compatibles d'une version à l'autre, alors que le code source a toujours conservé la compatibilité ascendante.

Each element of the sequences returned by `st2list()` or `st2tuple()` has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file `Include/graminit.h` and the Python module `symbol`. Each additional element of the sequence represents a component of the production as recognized in the input string : these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where 1 is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, 'if', 12)`, where the 12 represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module `token`.

The ST objects are not required to support the functionality of this module, but are provided for three purposes : to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple « wrapper » class may be created in Python to hide the use of ST objects.

The `parser` module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

Voir aussi :

Module `symbol` Useful constants representing internal nodes of the parse tree.

Module `token` Useful constants representing leaf nodes of the parse tree and functions for testing node values.

32.1.1 Creating ST Objects

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the 'eval' and 'exec' forms.

`parser.expr(source)`

The `expr()` function parses the parameter `source` as if it were an input to `compile(source, 'file.py', 'eval')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.suite(source)`

The `suite()` function parses the parameter `source` as if it were an input to `compile(source, 'file.py', 'exec')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.sequence2st(sequence)`

This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the caller. If there is a problem creating the internal representation, or if the tree cannot be validated, a `ParserError` exception is raised. An ST object created this way should not be assumed to compile correctly ; normal exceptions raised by compilation may still be initiated when the ST object is passed to `compilest()`. This may indicate problems not related to syntax (such as a `MemoryError` exception), but may also be due to constructs such as the result of parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, 'name')` or as three-element lists of the form `(1, 'name', 56)`. If the third element is present, it is assumed to be a valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

`parser.tuple2st(sequence)`

This is the same function as `sequence2st()`. This entry point is maintained for backward compatibility.

32.1.2 Converting ST Objects

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple-trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

`parser.st2list(st, line_info=False, col_info=False)`

This function accepts an ST object from the caller in `st` and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, `st2tuple()` should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If `line_info` is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token *ends*. This information is omitted if the flag is false or omitted.

`parser.st2tuple(st, line_info=False, col_info=False)`

This function accepts an ST object from the caller in `st` and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to `st2list()`.

If `line_info` is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

`parser.compilest(st, filename='<syntax-tree>')`

The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of a call to the built-in `exec()` or `eval()` functions. This function provides the interface to the compiler, passing the internal parse tree from `st` to the parser, using the source file name specified by the `filename` parameter. The default value supplied for `filename` indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a `SyntaxError` caused by the parse tree for `del f(0)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The `SyntaxError` raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the `parser` module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

32.1.3 Queries on ST Objects

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via `expr()` or `suite()` or from a parse tree via `sequence2st()`.

`parser.isexpr(st)`

When `st` represents an 'eval' form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by `compilest()` cannot be queried like this either, and are identical to those created by the built-in `compile()` function.

`parser.issuite(st)`

This function mirrors `isexpr()` in that it reports whether an ST object represents an 'exec' form, commonly known as a « suite. » It is not safe to assume that this function is equivalent to `not isexpr(st)`, as additional syntactic fragments may be supported in the future.

32.1.4 Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

exception `parser.ParserError`

Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in `SyntaxError` raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to `sequence2st()` and an explanatory string. Calls to `sequence2st()` need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions `compilest()`, `expr()`, and `suite()` may raise exceptions which are normally raised by the parsing and compilation process. These include the built in exceptions `MemoryError`, `OverflowError`, `SyntaxError`, and `SystemError`. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

32.1.5 ST Objects

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the `pickle` module) is also supported.

parser.STType

The type of the objects returned by `expr()`, `suite()` and `sequence2st()`.

ST objects have the following methods :

```
ST.compile(filename='<syntax-tree>')
    Same as compilest(st, filename).

ST.isexpr()
    Same as isexpr(st).

ST.issuite()
    Same as issuite(st).

ST.tolist(line_info=False, col_info=False)
    Same as st2list(st, line_info, col_info).

ST.totuple(line_info=False, col_info=False)
    Same as st2tuple(st, line_info, col_info).
```

32.1.6 Example : Emulation of `compile()`

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the `parser` module to produce an intermediate data structure is equivalent to the code

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the `parser` module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object :


```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions :

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

32.2 ast — Arbres Syntaxiques Abstraits

Code source : [Lib/ast.py](#)

Le module `ast` permet aux applications Python de traiter la grammaire abstraite de l'arbre syntaxique Python. La grammaire abstraite Python elle-même est susceptible d'être modifiée à chaque nouvelle version de Python ; ce module permet de trouver à quoi la grammaire actuelle ressemble.

Un arbre syntaxique abstrait peut être généré en passant l'option `ast.PyCF_ONLY_AST` à la fonction native `compile()`, ou en utilisant la fonction de facilité `parse()` fournie par le module. Le résultat est un arbre composé d'objets dont les classes héritent toutes de `ast.AST`. Un arbre syntaxique abstrait peut être compilé en code objet Python en utilisant la fonction native `compile()`.

32.2.1 Les classes nœud

`class ast.AST`

C'est la classe de base de toute classe nœud de l'AST. Les classes nœud courantes sont dérivées du fichier `Parser/Python.asdl`, qui est reproduit *ci-dessous*. Ils sont définis dans le module `C_ast` et ré-exportés dans le module `ast`.

Il y a une classe définie pour chacun des symboles présents à gauche dans la grammaire abstraite (par exemple, `ast.stmt` ou `ast.expr`). En plus de cela, il y a une classe définie pour chacun des constructeurs présentés à droite ; ces classes héritent des classes situées à gauche dans l'arbre. Par exemple, la classe `ast.BinOp` hérite de la classe `ast.expr`. Pour les règles de réécriture avec alternatives (comme *sums*), la partie gauche est abstraite : seules les instances des constructeurs spécifiques aux nœuds sont créés.

`_fields`

Chaque classe concrète possède un attribut `_fields` donnant les noms de tous les nœuds enfants.

Chaque instance d'une classe concrète possède un attribut pour chaque nœud enfant, du type défini par la grammaire. Par exemple, les instances `ast.BinOp` possèdent un attribut `left` de type `ast.expr`.

Si ces attributs sont marqués comme optionnels dans la grammaire (en utilisant un point d'interrogation ?), la valeur peut être `None`. Si les attributs peuvent avoir zéro ou plus valeurs (marqués avec un astérisque *), les valeurs sont représentées par des listes Python. Tous les attributs possibles doivent être présents et avoir une valeur valide pour compiler un AST avec `compile()`.

lineno
col_offset

Les instances des sous-classes `ast.expr` et `ast.stmt` possèdent les attributs `lineno` et `col_offset`. L'attribut `lineno` est le numéro de ligne dans le code source (indexé à partir de 1 tel que la première ligne est la ligne 1) et l'attribut `col_offset` qui représente le décalage UTF-8 en byte du premier jeton qui a généré le nœud. Le décalage UTF-8 est enregistré parce que l'analyseur syntaxique utilise l'UTF-8 en interne.

Le constructeur d'une classe `ast.T` analyse ses arguments comme suit :

- S'il y a des arguments positionnels, il doit y avoir autant de termes dans `T._fields` ; ils sont assignés comme attributs portant ces noms.
- S'il y a des arguments nommés, ils définissent les attributs de mêmes noms avec les valeurs données.

Par exemple, pour créer et peupler un nœud `ast.UnaryOp`, on peut utiliser

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

ou, plus compact

```
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                  lineno=0, col_offset=0)
```

32.2.2 Grammaire abstraite

La grammaire abstraite est actuellement définie comme suit :

```
-- ASDL's 7 builtin types are:
-- identifier, int, string, bytes, object, singleton, constant
--
-- singleton: None, True or False
-- constant can be None, whereas None means "no value" for object.

module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

    -- not really an actual node but useful in Jython's typesystem.
    | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                      stmt* body, expr* decorator_list, expr? returns)
        | AsyncFunctionDef(identifier name, arguments args,
                          stmt* body, expr* decorator_list, expr? returns)

    | ClassDef(identifier name,
              expr* bases,
              keyword* keywords,
              stmt* body,
              expr* decorator_list)
    | Return(expr? value)
```

(suite sur la page suivante)

(suite de la page précédente)

```

| Delete(expr* targets)
| Assign(expr* targets, expr value)
| AugAssign(expr target, operator op, expr value)
-- 'simple' indicates that we annotate simple name without parens
| AnnAssign(expr target, expr annotation, expr? value, int simple)

-- use 'orelse' because else is a keyword in target languages
| For(expr target, expr iter, stmt* body, stmt* orelse)
| AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
| While(expr test, stmt* body, stmt* orelse)
| If(expr test, stmt* body, stmt* orelse)
| With(withitem* items, stmt* body)
| AsyncWith(withitem* items, stmt* body)

| Raise(expr? exc, expr? cause)
| Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
| Assert(expr test, expr? msg)

| Import(alias* names)
| ImportFrom(identifiant? module, alias* names, int? level)

| Global(identifiant* names)
| Nonlocal(identifiant* names)
| Expr(expr value)
| Pass | Break | Continue

-- XXX Jython will be different
-- col_offset is the byte offset in the utf8 string the parser uses
attributes (int lineno, int col_offset)

-- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
| BinOp(expr left, operator op, expr right)
| UnaryOp(unaryop op, expr operand)
| Lambda(arguments args, expr body)
| IfExp(expr test, expr body, expr orelse)
| Dict(expr* keys, expr* values)
| Set(expr* elts)
| ListComp(expr elt, comprehension* generators)
| SetComp(expr elt, comprehension* generators)
| DictComp(expr key, expr value, comprehension* generators)
| GeneratorExp(expr elt, comprehension* generators)
-- the grammar constrains where yield expressions can occur
| Await(expr value)
| Yield(expr? value)
| YieldFrom(expr value)
-- need sequences for compare to distinguish between
-- x < 4 < 3 and (x < 4) < 3
| Compare(expr left, cmpop* ops, expr* comparators)
| Call(expr func, expr* args, keyword* keywords)
| Num(object n) -- a number as a PyObject.
| Str(string s) -- need to specify raw, unicode, etc?
| FormattedValue(expr value, int? conversion, expr? format_spec)
| JoinedStr(expr* values)
| Bytes(bytes s)
| NameConstant(singleton value)

```

(suite sur la page suivante)

```

    | Ellipsis
    | Constant(constant value)

    -- the following expression can appear in assignment context
    | Attribute(expr value, identifier attr, expr_context ctx)
    | Subscript(expr value, slice slice, expr_context ctx)
    | Starred(expr value, expr_context ctx)
    | Name(identifier id, expr_context ctx)
    | List(expr* elts, expr_context ctx)
    | Tuple(expr* elts, expr_context ctx)

    -- col_offset is the byte offset in the utf8 string the parser uses
    attributes (int lineno, int col_offset)

expr_context = Load | Store | Del | AugLoad | AugStore | Param

slice = Slice(expr? lower, expr? upper, expr? step)
        | ExtSlice(slice* dims)
        | Index(expr value)

boolop = And | Or

operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
          | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs, int is_async)

excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
               attributes (int lineno, int col_offset)

arguments = (arg* args, arg? vararg, arg* kwonlyargs, expr* kw_defaults,
            arg? kwarg, expr* defaults)

arg = (identifier arg, expr? annotation)
      attributes (int lineno, int col_offset)

-- keyword arguments supplied to call (NULL identifier for **kwargs)
keyword = (identifier? arg, expr value)

-- import name with optional 'as' alias.
alias = (identifier name, identifier? asname)

withitem = (expr context_expr, expr? optional_vars)
}

```

32.2.3 Outils du module `ast`

À part la classe nœud, le module `ast` définit ces fonctions et classes utilitaires pour traverser les arbres syntaxiques abstraits :

`ast.parse(source, filename='<unknown>', mode='exec')`
 Analyse le code source en un nœud AST. Équivalent à `compile(source, filename, mode, ast.PyCF_ONLY_AST)`.

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

`ast.literal_eval(node_or_string)`
 Évalue de manière sûre un nœud expression ou une chaîne de caractères contenant une expression littérale Python ou un conteneur. La chaîne de caractères ou le nœud fourni peut seulement faire partie des littéraux Python suivants : chaînes de caractères, bytes, nombres, n-uplets, listes, dictionnaires, ensembles, booléens, et `None`.
 Cela peut être utilisé pour évaluer de manière sûre la chaîne de caractères contenant des valeurs Python de sources non fiable sans avoir besoin d'analyser les valeurs elles-mêmes. Cette fonction n'est pas capable d'évaluer des expressions complexes arbitraires, par exemple impliquant des opérateurs ou de l'indexation.

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

Modifié dans la version 3.2 : Accepte maintenant les littéraux suivants *bytes* et *sets*.

`ast.get_docstring(node, clean=True)`
 Return the docstring of the given *node* (which must be a `FunctionDef`, `ClassDef` or `Module` node), or `None` if it has no docstring. If *clean* is true, clean up the docstring's indentation with `inspect.cleandoc()`.

`ast.fix_missing_locations(node)`
 Lorsque l'on compile un arbre avec `compile()`, le compilateur attend les attributs `lineno` et `col_offset` pour tous les nœuds qui les supportent. Il est fastidieux de les remplir pour les nœuds générés, cette fonction utilitaire ajoute ces attributs de manière récursive là où ils ne sont pas déjà définis, en les définissant comme les valeurs du nœud parent. Elle fonctionne récursivement en démarrant de *node*.

`ast.increment_lineno(node, n=1)`
 Incrmente de *n* le numéro de ligne de chaque nœud dans l'arbre en commençant par le nœud *node*. C'est utile pour « déplacer du code » à un endroit différent dans un fichier.

`ast.copy_location(new_node, old_node)`
 Copie le code source (`lineno` et `col_offset`) de l'ancien nœud *old_node* vers le nouveau nœud *new_node* si possible, et renvoie *new_node*.

`ast.iter_fields(node)`
 Produit un n-uplet de (*fieldname*, *value*) pour chaque champ de *node._fields* qui est présent dans *node*.

`ast.iter_child_nodes(node)`
 Produit tous les nœuds enfants directs de *node*, c'est à dire, tous les champs qui sont des nœuds et tous les éléments des champs qui sont des listes de nœuds.

`ast.walk(node)`
 Produit récursivement tous les nœuds enfants dans l'arbre en commençant par *node* (*node* lui-même est inclus), sans ordre spécifique. C'est utile lorsque l'on souhaite modifier les nœuds sur place sans prêter attention au contexte.

class `ast.NodeVisitor`

Classe de base pour un visiteur de nœud, qui parcourt l'arbre syntaxique abstrait et appelle une fonction de visite

pour chacun des nœuds trouvés. Cette fonction peut renvoyer une valeur qui est transmise par la méthode `visit()`. Cette classe est faite pour être dérivée, en ajoutant des méthodes de visite à la sous-classe.

visit (*node*)

Visite un nœud. L'implémentation par défaut appelle la méthode `self.visit_classname` où *classname* représente le nom de la classe du nœud, ou `generic_visit()` si cette méthode n'existe pas.

generic_visit (*node*)

Le visiteur appelle la méthode `visit()` de tous les enfants du nœud.

Notons que les nœuds enfants qui possèdent une méthode de visite spéciale ne seront pas visités à moins que le visiteur n'appelle la méthode `generic_visit()` ou ne les visite lui-même.

N'utilisez pas `NodeVisitor` si vous souhaitez appliquer des changements sur les nœuds lors du parcours. Pour cela, un visiteur spécial existe (`NodeTransformer`) qui permet les modifications.

class `ast.NodeTransformer`

Une sous-classe `NodeVisitor` qui traverse l'arbre syntaxique abstrait et permet les modifications des nœuds.

Le `NodeTransformer` traverse l'AST et utilise la valeur renvoyée par les méthodes du visiteur pour remplacer ou supprimer l'ancien nœud. Si la valeur renvoyée par la méthode du visiteur est `None`, le nœud est supprimé de sa position, sinon il est remplacé par la valeur de retour. La valeur de retour peut être le nœud original et dans ce cas, il n'y a pas de remplacement.

Voici un exemple du *transformer* qui réécrit les occurrences du dictionnaire (`foo`) en `data['foo']` :

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return copy_location(Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        ), node)
```

Gardez en tête que si un nœud sur lequel vous travaillez a des nœuds enfants, vous devez transformer également ces nœuds enfant vous-même ou appeler d'abord la méthode `generic_visit()` sur le nœud.

Pour les nœuds qui font partie d'une collection d'instructions (cela s'applique à tous les nœuds instruction), le visiteur peut aussi renvoyer la liste des nœuds plutôt qu'un seul nœud.

Utilisation typique du *transformer* :

```
node = YourTransformer().visit(node)
```

`ast.dump` (*node*, *annotate_fields=True*, *include_attributes=False*)

Renvoie un *dump* formaté de l'arbre dans *node*. C'est principalement utile à des fins de débogage. La chaîne de caractères renvoyée présente les noms et valeurs des champs. Cela rend le code impossible à évaluer, si l'on souhaite évaluer ce code, l'option *annotate_fields* doit être définie comme `False`. Les attributs comme les numéros de ligne et les décalages de colonne ne sont pas récupérés par défaut. Si l'on souhaite les récupérer, l'option *include_attributes* peut être définie comme `True`.

Voir aussi :

[Green Tree Snakes](#), an external documentation resource, has good details on working with Python ASTs.

32.3 `symtable` — Access to the compiler’s symbol tables

Code source : [Lib/symtable.py](#)

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. `symtable` provides an interface to examine these tables.

32.3.1 Generating Symbol Tables

`symtable.symtable (code, filename, compile_type)`

Return the toplevel `SymbolTable` for the Python source `code`. `filename` is the name of the file containing the code. `compile_type` is like the `mode` argument to `compile()`.

32.3.2 Examining Symbol Tables

class `symtable.SymbolTable`

A namespace table for a block. The constructor is not public.

get_type()

Return the type of the symbol table. Possible values are 'class', 'module', and 'function'.

get_id()

Return the table’s identifier.

get_name()

Return the table’s name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or 'top' if the table is global (`get_type()` returns 'module').

get_lineno()

Return the number of the first line in the block this table represents.

is_optimized()

Return True if the locals in this table can be optimized.

is_nested()

Return True if the block is a nested class or function.

has_children()

Return True if the block has nested namespaces within it. These can be obtained with `get_children()`.

has_exec()

Return True if the block uses `exec`.

get_identifiers()

Return a list of names of symbols in this table.

lookup (name)

Lookup `name` in the table and return a `Symbol` instance.

get_symbols()

Return a list of `Symbol` instances for names in the table.

get_children()

Return a list of the nested symbol tables.

class `symtable.Function`

A namespace for a function or method. This class inherits `SymbolTable`.

get_parameters()

Return a tuple containing names of parameters to this function.

get_locals()

Return a tuple containing names of locals in this function.

get_globals()
Return a tuple containing names of globals in this function.

get_frees()
Return a tuple containing names of free variables in this function.

class `symtable.Class`
A namespace of a class. This class inherits `SymbolTable`.

get_methods()
Return a tuple containing the names of methods declared in the class.

class `symtable.Symbol`
An entry in a `SymbolTable` corresponding to an identifier in the source. The constructor is not public.

get_name()
Return the symbol's name.

is_referenced()
Return `True` if the symbol is used in its block.

is_imported()
Return `True` if the symbol is created from an import statement.

is_parameter()
Return `True` if the symbol is a parameter.

is_global()
Return `True` if the symbol is global.

is_declared_global()
Return `True` if the symbol is declared global with a global statement.

is_local()
Return `True` if the symbol is local to its block.

is_free()
Return `True` if the symbol is referenced in its block, but not assigned to.

is_assigned()
Return `True` if the symbol is assigned to in its block.

is_namespace()
Return `True` if name binding introduces new namespace.
If the name is used as the target of a function or class statement, this will be true.
Par exemple :

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

get_namespaces()
Return a list of namespaces bound to this name.

get_namespace()
Return the namespace bound to this name. If more than one namespace is bound, `ValueError` is raised.

32.4 `symbol` — Constantes utilisées dans les Arbres Syntaxiques

Code source : [Lib/symbol.py](#)

Ce module fournit des constantes représentant les valeurs numériques des nœuds internes de l'analyseur. Contrairement à la plupart des constantes en Python, celles-ci utilisent des noms en minuscules. Référez-vous au fichier `Grammar/Grammar` dans la distribution de Python pour les définitions de ces noms dans le contexte de la grammaire du langage. La valeur numérique correspondant au nom peut changer d'une version de Python à l'autre.

Ce module fournit aussi ces objets :

`symbol.sym_name`

Dictionnaire faisant correspondre les valeurs numériques des constantes définies dans ce module à leurs noms, permettant de générer une représentation plus humaine des arbres syntaxiques.

32.5 `token` — Constantes utilisées avec les arbres d'analyse Python (*parse trees*)

Code source : [Lib/token.py](#)

Ce module fournit des constantes qui représentent les valeurs numériques des nœuds enfants du *parse tree* (les jetons « terminaux »). Voir le fichier `Grammar/Grammar` dans la distribution Python pour la définitions des noms dans le contexte de la grammaire. Les valeurs numériques correspondant aux noms sont susceptibles de changer entre deux versions de Python.

Le module fournit également une correspondance entre les codes numériques et les noms et certaines fonctions. Les fonctions reflètent les définitions des fichiers d'en-tête C de Python.

`token.tok_name`

Dictionnaire faisant correspondre les valeurs numériques des constantes définies dans ce module à leurs noms, permettant de générer une représentation plus humaine des arbres syntaxiques.

`token.ISTERMINAL(x)`

Renvoie `True` pour les valeurs des jetons terminaux.

`token.ISNONTERMINAL(x)`

Renvoie `True` pour les valeurs des jetons non terminaux.

`token.ISEOF(x)`

Renvoie `True` si `x` est le marqueur indiquant la fin de la source.

Les constantes associées aux jetons sont :

`token.ENDMARKER`

`token.NAME`

`token.NUMBER`

`token.STRING`

`token.NEWLINE`

`token.INDENT`

`token.DEDENT`

`token.LPAR`

`token.RPAR`

`token.LSQB`

`token.RSQB`
`token.COLON`
`token.COMMA`
`token.SEMI`
`token.PLUS`
`token.MINUS`
`token.STAR`
`token.SLASH`
`token.VBAR`
`token.AMPER`
`token.LESS`
`token.GREATER`
`token.EQUAL`
`token.DOT`
`token.PERCENT`
`token.LBRACE`
`token.RBRACE`
`token.EQEQUAL`
`token.NOTEQUAL`
`token.LESSEQUAL`
`token.GREATEREQUAL`
`token.TILDE`
`token.CIRCUMFLEX`
`token.LEFTSHIFT`
`token.RIGHTSHIFT`
`token.DOUBLESTAR`
`token.PLUSEQUAL`
`token.MINEQUAL`
`token.STAREQUAL`
`token.SLASHEQUAL`
`token.PERCENTEQUAL`
`token.AMPEREQUAL`
`token.VBAREQUAL`
`token.CIRCUMFLEXEQUAL`
`token.LEFTSHIFTEQUAL`
`token.RIGHTSHIFTEQUAL`
`token.DOUBLESTAREQUAL`
`token.DOUBLESASH`
`token.DOUBLESASHEQUAL`
`token.AT`
`token.ATEQUAL`
`token.RARROW`
`token.ELLIPSIS`
`token.OP`
`token.AWAIT`
`token.ASYNC`
`token.ERRORTOKEN`
`token.N_TOKENS`
`token.NT_OFFSET`

Modifié dans la version 3.5 : Added *AWAIT* and *ASYNC* tokens. Starting with Python 3.7, « *async* » and « *await* » will be tokenized as *NAME* tokens, and *AWAIT* and *ASYNC* will be removed.

32.6 keyword — Tester si des chaînes sont des mot-clés Python

Code source : [Lib/keyword.py](#)

Ce module permet à un programme Python de déterminer si une chaîne représente un mot-clé.

`keyword.iskeyword(s)`

Renvoie vrai si *s* est un mot-clé Python.

`keyword.kwlist`

Séquence contenant tous les mots-clés définis par l'interpréteur. Si certains mots-clés sont définis pour être actifs seulement si des instructions `__future__` particulières sont effectives, ils seront tout de même inclus.

32.7 tokenize — Analyseur lexical de Python

Code source : [Lib/tokenize.py](#)

Le module `tokenize` fournit un analyseur lexical pour Python, implémenté en Python. L'analyseur de ce module renvoie les commentaires sous forme de *token*, ce qui le rend intéressant pour implémenter des *pretty-printers*, typiquement pour faire de la coloration syntaxique.

Pour simplifier la gestion de flux de *tokens*, tous les *tokens* operator et delimiter, ainsi que les Ellipsis* sont renvoyés en utilisant le *token* générique *OP*. Le type exact peut être déterminé en vérifiant la propriété `exact_type` du *named tuple* renvoyé par `tokenize.tokenize()`.

32.7.1 Analyse Lexicale

Le point d'entrée principal est un *générateur* :

`tokenize.tokenize(readline)`

Le générateur `tokenize()` prend un argument *readline* qui doit être un objet callable exposant la même interface que la méthode `io.IOBase.readline()` des objets fichiers. Chaque appel à la fonction doit renvoyer une ligne sous forme de *bytes*.

Le générateur fournit des quintuplet contenant : le type du *token*, sa chaîne, un couple d'entiers (*srow*, *scol*) indiquant la ligne et la colonne où le *token* commence, un couple d'entiers (*erow*, *ecol*) indiquant la ligne et la colonne où il se termine, puis la ligne dans laquelle il a été trouvé. La ligne donnée (le dernier élément du *tuple*) est la ligne « logique », les *continuation lines* étant incluses. Le *tuple* est renvoyé sous forme de *named tuple* dont les noms sont : `type string start end line`.

The returned *named tuple* has an additional property named `exact_type` that contains the exact operator type for *token.OP* tokens. For all other token types `exact_type` equals the named tuple type field.

Modifié dans la version 3.1 : Soutien ajouté pour *tuples* nommé.

Modifié dans la version 3.3 : Soutien ajouté pour `exact_type`.

`tokenize()` détermine le codage source du fichier en recherchant une nomenclature UTF-8 ou un cookie d'encodage, selon la [PEP 263](#).

All constants from the *token* module are also exported from *tokenize*, as are three additional token type values :

`tokenize.COMMENT`

Valeur du jeton utilisée pour indiquer un commentaire.

`tokenize.NL`

Token value used to indicate a non-terminating newline. The NEWLINE token indicates the end of a logical line of Python code ; NL tokens are generated when a logical line of code is continued over multiple physical lines.

`tokenize.ENCODING`

Token value that indicates the encoding used to decode the source bytes into text. The first token returned by `tokenize()` will always be an ENCODING token.

Une autre fonction est fournie pour inverser le processus de tokenisation. Ceci est utile pour créer des outils permettant de coder un script, de modifier le flux de jetons et de réécrire le script modifié.

`tokenize.untokenize(iterable)`

Convertit les jetons en code source Python. L'*iterable* doit renvoyer des séquences avec au moins deux éléments, le type de jeton et la chaîne de caractères associée. Tout élément de séquence supplémentaire est ignoré.

Le script reconstruit est renvoyé sous la forme d'une chaîne unique. Le résultat est garanti pour que le jeton corresponde à l'entrée afin que la conversion soit sans perte et que les allers et retours soient assurés. La garantie ne s'applique qu'au type de jeton et à la chaîne de jetons car l'espacement entre les jetons (positions des colonnes) peut changer.

It returns bytes, encoded using the ENCODING token, which is the first token sequence output by `tokenize()`.

`tokenize()` a besoin de détecter le codage des fichiers sources qu'il code. La fonction utilisée pour cela est disponible :

`tokenize.detect_encoding(readline)`

La fonction `detect_encoding()` est utilisée pour détecter l'encodage à utiliser pour décoder un fichier source Python. Il nécessite un seul argument, *readline*, de la même manière que le générateur `tokenize()`.

Il appelle *readline* au maximum deux fois et renvoie le codage utilisé (sous forme de chaîne) et une liste de toutes les lignes (non décodées à partir des octets) dans lesquelles il a été lu.

It detects the encoding from the presence of a UTF-8 BOM or an encoding cookie as specified in [PEP 263](#). If both a BOM and a cookie are present, but disagree, a `SyntaxError` will be raised. Note that if the BOM is found, `'utf-8-sig'` will be returned as an encoding.

Si aucun codage n'est spécifié, la valeur par défaut, `'utf-8'`, sera renvoyée.

Utilisez `open()` pour ouvrir les fichiers source Python : ça utilise `detect_encoding()` pour détecter le codage du fichier.

`tokenize.open(filename)`

Ouvre un fichier en mode lecture seule en utilisant l'encodage détecté par `detect_encoding()`.

Nouveau dans la version 3.2.

exception `tokenize.TokenError`

Déclenché lorsque soit une *docstring* soit une expression qui pourrait être divisée sur plusieurs lignes n'est pas complété dans le fichier, par exemple :

```
"""Beginning of
docstring
```

ou :

```
[1,
 2,
 3
```

Note that unclosed single-quoted strings do not cause an error to be raised. They are tokenized as `ERRORTOKEN`, followed by the tokenization of their contents.

32.7.2 Utilisation en ligne de commande.

Nouveau dans la version 3.3.

Le module `tokenize` peut être exécuté en tant que script à partir de la ligne de commande. C'est aussi simple que :

```
python -m tokenize [-e] [filename.py]
```

Les options suivantes sont acceptées :

-h, --help

Montre ce message d'aide et quitte

-e, --exact

Affiche les noms de jetons en utilisant le même type.

Si `filename.py` est spécifié, son contenu est tokenisé vers *stdout*. Sinon, la tokenisation est effectuée sur ce qui est fourni sur *stdin*.

32.7.3 Exemples

Exemple d'un script qui transforme les littéraux de type *float* en type *Decimal* :

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows). Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s) #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline) # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval: # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')])
```

(suite sur la page suivante)

(suite de la page précédente)

```

    })
    else:
        result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')

```

Exemple de tokenisation à partir de la ligne de commande. Le script :

```

def say_hello():
    print("Hello, World!")

say_hello()

```

sera tokenisé à la sortie suivante où la première colonne est la plage des coordonnées de la ligne/colonne où se trouve le jeton, la deuxième colonne est le nom du jeton, et la dernière colonne est la valeur du jeton (le cas échéant)

```

$ python -m tokenize hello.py
0,0-0,0:      ENCODING      'utf-8'
1,0-1,3:      NAME          'def'
1,4-1,13:     NAME          'say_hello'
1,13-1,14:    OP            '('
1,14-1,15:    OP            ')'
1,15-1,16:    OP            ':'
1,16-1,17:    NEWLINE      '\n'
2,0-2,4:      INDENT       '    '
2,4-2,9:      NAME          'print'
2,9-2,10:     OP            '('
2,10-2,25:    STRING        '"Hello, World!'"
2,25-2,26:    OP            ')'
2,26-2,27:    NEWLINE      '\n'
3,0-3,1:      NL           '\n'
4,0-4,0:      DEDENT       ''
4,0-4,9:      NAME          'say_hello'
4,9-4,10:     OP            '('
4,10-4,11:    OP            ')'
4,11-4,12:    NEWLINE      '\n'
5,0-5,0:      ENDMARKER    ''

```

The exact token type names can be displayed using the `-e` option :

```

$ python -m tokenize -e hello.py
0,0-0,0:      ENCODING      'utf-8'
1,0-1,3:      NAME          'def'
1,4-1,13:     NAME          'say_hello'
1,13-1,14:    LPAR         '('
1,14-1,15:    RPAR         ')'
1,15-1,16:    COLON        ':'
1,16-1,17:    NEWLINE      '\n'
2,0-2,4:      INDENT       '    '
2,4-2,9:      NAME          'print'
2,9-2,10:     LPAR         '('
2,10-2,25:    STRING        '"Hello, World!'"
2,25-2,26:    RPAR         ')'
2,26-2,27:    NEWLINE      '\n'
3,0-3,1:      NL           '\n'
4,0-4,0:      DEDENT       ''
4,0-4,9:      NAME          'say_hello'

```

(suite sur la page suivante)

(suite de la page précédente)

4, 9-4, 10:	LPAR	' ('
4, 10-4, 11:	RPAR	') '
4, 11-4, 12:	NEWLINE	' \n '
5, 0-5, 0:	ENDMARKER	' '

32.8 `tabnanny` — Détection d'indentation ambiguë

Code source : [Lib/tabnanny.py](#)

Pour l'instant ce module est destiné à être appelé comme un script. Toutefois, il est possible de l'importer dans un IDE et d'utiliser la fonction `check()` décrite ci-dessous.

Note : L'API fournie par ce module est susceptible de changer dans les versions futures ; ces modifications peuvent ne pas être rétro-compatibles.

`tabnanny.check(file_or_dir)`

Si `file_or_dir` est un répertoire et non un lien symbolique, alors descend récursivement l'arborescence de répertoire nommé par `file_or_dir`, en vérifiant tous les fichiers `.py` en chemin. Si `file_or_dir` est un fichier source Python ordinaire, il est vérifié pour les problèmes liés aux espaces blancs. Les messages de diagnostic sont écrits sur la sortie standard à l'aide de la fonction `print()`.

`tabnanny.verbose`

Option indiquant s'il faut afficher des messages détaillés. Cela est incrémenté par l'option `-v` s'il est appelé comme un script.

`tabnanny.filename_only`

Option indiquant s'il faut afficher uniquement les noms de fichiers contenant des problèmes liés aux espaces blancs. Est défini à `True` par l'option `-q` s'il est appelé comme un script.

exception `tabnanny.NannyNag`

Déclenché par `process_tokens()` si une indentation ambiguë est détectée. Capturé et géré dans `check()`.

`tabnanny.process_tokens(tokens)`

Cette fonction est utilisée par `check()` pour traiter les jetons générés par le module `tokenize`.

Voir aussi :

Module `tokenize` Analyseur lexical pour le code source Python.

32.9 `pyclbr` — Python class browser support

Code source : [Lib/pyclbr.py](#)

The `pyclbr` module can be used to determine some limited information about the classes, methods and top-level functions defined in a module. The information provided is sufficient to implement a traditional three-pane class browser. The information is extracted from the source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

`pyclbr.readmodule(module, path=None)`

Read a module and return a dictionary mapping class names to class descriptor objects. The parameter *module* should be the name of a module as a string; it may be the name of a module within a package. The *path* parameter should be a sequence, and is used to augment the value of `sys.path`, which is used to locate module source code.

`pyclbr.readmodule_ex(module, path=None)`

Like `readmodule()`, but the returned dictionary, in addition to mapping class names to class descriptor objects, also maps top-level function names to function descriptor objects. Moreover, if the module being read is a package, the key `'__path__'` in the returned dictionary has as its value a list which contains the package search path.

32.9.1 Objets classes

The `Class` objects used as values in the dictionary returned by `readmodule()` and `readmodule_ex()` provide the following data attributes :

`Class.module`

The name of the module defining the class described by the class descriptor.

`Class.name`

The name of the class.

`Class.super`

A list of `Class` objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by `readmodule()` are listed as a string with the class name instead of as `Class` objects.

`Class.methods`

A dictionary mapping method names to line numbers.

`Class.file`

Name of the file containing the `class` statement defining the class.

`Class.lineno`

The line number of the `class` statement within the file named by *file*.

32.9.2 Objets fonctions

The `Function` objects used as values in the dictionary returned by `readmodule_ex()` provide the following attributes :

`Function.module`

The name of the module defining the function described by the function descriptor.

`Function.name`

The name of the function.

`Function.file`

Name of the file containing the `def` statement defining the function.

`Function.lineno`

The line number of the `def` statement within the file named by *file*.

32.10 `py_compile` — Compile Python source files

Code source : `Lib/py_compile.py`

The `py_compile` module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

exception `py_compile.PyCompileError`

Exception raised when an error occurs while attempting to compile the file.

`py_compile.compile(file, cfile=None, dfile=None, doraise=False, optimize=-1)`

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named *file*. The byte-code is written to *cfile*, which defaults to the [PEP 3147/PEP 488](#) path, ending in `.pyc`. For example, if *file* is `/foo/bar/baz.py` *cfile* will default to `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. If *dfile* is specified, it is used as the name of the source file in error messages when instead of *file*. If *doraise* is true, a `PyCompileError` is raised when an error is encountered while compiling *file*. If *doraise* is false (the default), an error string is written to `sys.stderr`, but no exception is raised. This function returns the path to byte-compiled file, i.e. whatever *cfile* value was used.

If the path that *cfile* becomes (either explicitly specified or computed) is a symlink or non-regular file, `FileExistsError` will be raised. This is to act as a warning that import will turn those paths into regular files if it is allowed to write byte-compiled files to those paths. This is a side-effect of import using file renaming to place the final byte-compiled file into place to prevent concurrent file writing issues.

optimize controls the optimization level and is passed to the built-in `compile()` function. The default of `-1` selects the optimization level of the current interpreter.

Modifié dans la version 3.2 : Changed default value of *cfile* to be [PEP 3147](#)-compliant. Previous default was *file* + `'c'` (`'o'` if optimization was enabled). Also added the *optimize* parameter.

Modifié dans la version 3.4 : Changed code to use `importlib` for the byte-code cache file writing. This means file creation/writing semantics now match what `importlib` does, e.g. permissions, write-and-move semantics, etc. Also added the caveat that `FileExistsError` is raised if *cfile* is a symlink or non-regular file.

`py_compile.main(args=None)`

Compile several source files. The files named in *args* (or on the command line, if *args* is `None`) are compiled and the resulting byte-code is cached in the normal manner. This function does not search a directory structure to locate source files; it only compiles files named explicitly. If `'-'` is the only parameter in *args*, the list of files is taken from standard input.

Modifié dans la version 3.2 : Added support for `'-'`.

When this module is run as a script, the `main()` is used to compile all the files named on the command line. The exit status is nonzero if one of the files could not be compiled.

Voir aussi :

Module `compileall` Utilities to compile all Python source files in a directory tree.

32.11 `compileall` — Byte-compile Python libraries

Code source : [Lib/compileall.py](#)

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don't have write permission to the library directories.

32.11.1 Command-line use

This module can work as a script (using `python -m compileall`) to compile Python sources.

directory ...

file ...

Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was `-l <directories from sys.path>`.

-l

Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

-f

Force rebuild even if timestamps are up-to-date.

-q

Do not print the list of files compiled. If passed once, error messages will still be printed. If passed twice (`-qq`), all output is suppressed.

-d `destdir`

Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

-x `regex`

`regex` is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

-i `list`

Read the file `list` and add each line that it contains to the list of files and directories to compile. If `list` is `-`, read lines from `stdin`.

-b

Write the byte-code files to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

-r

Control the maximum recursion level for subdirectories. If this is given, then `-l` option will not be taken into account. `python -m compileall <directory> -r 0` is equivalent to `python -m compileall <directory> -l`.

-j `N`

Use `N` workers to compile the files within the given directory. If 0 is used, then the result of `os.cpu_count()` will be used.

Modifié dans la version 3.2 : Added the `-i`, `-b` and `-h` options.

Modifié dans la version 3.5 : Added the `-j`, `-r`, and `-qq` options. `-q` option was changed to a multilevel value. `-b` will always produce a byte-code file ending in `.pyc`, never `.pyo`.

There is no command-line option to control the optimization level used by the `compile()` function, because the Python interpreter itself already provides the option : `python -O -m compileall`.

32.11.2 Public functions

`compileall.compile_dir` (*dir*, *maxlevels*=10, *ddir*=None, *force*=False, *rx*=None, *quiet*=0, *legacy*=False, *optimize*=-1, *workers*=1)

Recursively descend the directory tree named by *dir*, compiling all `.py` files along the way. Return a true value if all the files compiled successfully, and a false value otherwise.

The *maxlevels* parameter is used to limit the depth of the recursion; it defaults to 10.

If *ddir* is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *force* is true, modules are re-compiled even if the timestamps are up to date.

If *rx* is given, its search method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped.

If *quiet* is False or 0 (the default), the filenames and other information are printed to standard out. Set to 1, only errors are printed. Set to 2, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their [PEP 3147](#) locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in `compile()` function.

The argument *workers* specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can't use multiple workers and *workers* argument is given, then sequential compilation will be used as a fallback. If *workers* is lower than 0, a `ValueError` will be raised.

Modifié dans la version 3.2 : Added the *legacy* and *optimize* parameter.

Modifié dans la version 3.5 : Added the *workers* parameter.

Modifié dans la version 3.5 : *quiet* parameter was changed to a multilevel value.

Modifié dans la version 3.5 : The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`compileall.compile_file` (*fullname*, *ddir*=None, *force*=False, *rx*=None, *quiet*=0, *legacy*=False, *optimize*=-1)

Compile the file with path *fullname*. Return a true value if the file compiled successfully, and a false value otherwise.

If *ddir* is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *rx* is given, its search method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and `True` is returned.

If *quiet* is False or 0 (the default), the filenames and other information are printed to standard out. Set to 1, only errors are printed. Set to 2, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their [PEP 3147](#) locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in `compile()` function.

Nouveau dans la version 3.2.

Modifié dans la version 3.5 : *quiet* parameter was changed to a multilevel value.

Modifié dans la version 3.5 : The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

`compileall.compile_path(skip_curodir=True, maxlevels=0, force=False, quiet=0, legacy=False, optimize=-1)`

Byte-compile all the `.py` files found along `sys.path`. Return a true value if all the files compiled successfully, and a false value otherwise.

If *skip_curodir* is true (the default), the current directory is not included in the search. All other parameters are passed to the `compile_dir()` function. Note that unlike the other compile functions, *maxlevels* defaults to 0.

Modifié dans la version 3.2 : Added the *legacy* and *optimize* parameter.

Modifié dans la version 3.5 : *quiet* parameter was changed to a multilevel value.

Modifié dans la version 3.5 : The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

To force a recompile of all the `.py` files in the `Lib/` subdirectory and all its subdirectories :

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\](.)svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

Voir aussi :

Module `py_compile` Byte-compile a single source file.

32.12 `dis` – Désassembleur pour le code intermédiaire de Python

Code source : [Lib/dis.py](#)

La bibliothèque `dis` supporte l'analyse du *bytecode* CPython en le désassemblant. Le code intermédiaire CPython, que cette bibliothèque prend en paramètre, est défini dans le fichier `Include/opcode.h` et est utilisé par le compilateur et l'interpréteur.

CPython implementation detail : Le code intermédiaire est un détail d'implémentation de l'interpréteur CPython. Il n'y a pas de garantie que le code intermédiaire sera ajouté, retiré, ou modifié dans les différentes versions de Python. L'utilisation de cette bibliothèque ne fonctionne pas nécessairement sur les machines virtuelles Python ni les différentes versions de Python.

Modifié dans la version 3.6 : Utilisez 2 bits pour chaque instruction. Avant, le nombre de bits variait par instruction.

Exemple : Etant donné la fonction `myfunc()` :

```
def myfunc(alist):
    return len(alist)
```

la commande suivante peut-être utilisé pour afficher le désassemblage de `myfunc()` :

```
>>> dis.dis(myfunc)
2          0 LOAD_GLOBAL           0 (len)
          2 LOAD_FAST             0 (alist)
          4 CALL_FUNCTION         1
          6 RETURN_VALUE
```

(Le « 2 » est un numéro de ligne).

32.12.1 Analyse du code intermédiaire

Nouveau dans la version 3.4.

L'analyse de l'API code intermédiaire permet de rassembler des blocs de code en Python dans une classe `Bytecode`, qui permet un accès facile aux détails du code compilé.

class `dis.Bytecode` (*x*, *, *first_line=None*, *current_offset=None*)

Analyse the bytecode corresponding to a function, generator, method, string of source code, or a code object (as returned by `compile()`).

Ceci est *wrapper* sur plusieurs fonctions de la liste ci-dessous, notamment `get_instructions()`, étant donné qu'une itération sur une instance de la classe `Bytecode` rend les opérations du code intermédiaire des instances de `Instruction`.

Si *first_line* ne vaut pas `None`, elle indique le nombre de la ligne qui doit être considérée comme première ligne source dans le code désassemblé. Autrement, les informations sur la ligne source sont prises directement à partir de la classe du code désassemblé.

Si la valeur de *current_offset* est différente de `None`, c'est une référence à un offset d'une instruction dans le code désassemblé. Cela veut dire que `dis()` va générer un marqueur de « l'instruction en cours » contre le code d'opération donné.

classmethod `from_traceback` (*tb*)

Construisez une instance `Bytecode` à partir de la trace d'appel, en mettant *current_offset* à l'instruction responsable de l'exception.

codeobj

Le code compilé objet.

first_line

La première ligne source du code objet (si disponible)

dis()

Retourne une vue formatée des opérations du code intermédiaire (la même que celle envoyée par `dis.dis()`, mais comme une chaîne de caractères de plusieurs lignes).

info()

Retourne une chaîne de caractères de plusieurs lignes formatée avec des informations détaillées sur l'objet code comme `code_info()`.

Exemple :

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```

32.12.2 Analyse de fonctions

La bibliothèque `dis` comprend également l'analyse des fonctions suivantes, qui envoient l'entrée directement à la sortie souhaitée. Elles peuvent être utiles si il n'y a qu'une seule opération à effectuer, la représentation intermédiaire objet n'étant donc pas utile dans ce cas :

`dis.code_info(x)`

Return a formatted multi-line string with detailed code object information for the supplied function, generator, method, source code string or code object.

Il est à noter que le contenu exact des chaînes de caractères figurant dans les informations du code dépendent fortement sur l'implémentation, et peuvent changer arbitrairement sous machines virtuelles Python ou les versions de Python.

Nouveau dans la version 3.2.

`dis.show_code(x, *, file=None)`

Affiche des informations détaillées sur le code de la fonction fournie, la méthode, la chaîne de caractère du code source ou du code objet à `file` (ou bien `sys.stdout` si `file` n'est pas spécifié).

Ceci est un raccourci convenable de `print(code_info(x), file=file)`, principalement fait pour l'exploration interactive sur l'invite de l'interpréteur.

Nouveau dans la version 3.2.

Modifié dans la version 3.4 : Ajout du paramètre `file`.

`dis.dis(x=None, *, file=None)`

Disassemble the `x` object. `x` can denote either a module, a class, a method, a function, a generator, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. Strings are first compiled to code objects with the `compile()` built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

Le désassemblage est envoyé sous forme de texte à l'argument du fichier `file` si il est fourni, et à `sys.stdout` sinon.

Modifié dans la version 3.4 : Ajout du paramètre `file`.

`dis.distb(tb=None, *, file=None)`

Désassemble la fonction du haut de la pile des traces d'appels, en utilisant la dernière trace d'appels si rien n'a été envoyé. L'instruction à l'origine de l'exception est indiquée.

Le désassemblage est envoyé sous forme de texte à l'argument du fichier `file` si il est fourni, et à `sys.stdout` sinon.

Modifié dans la version 3.4 : Ajout du paramètre `file`.

`dis.disassemble(code, lasti=-1, *, file=None)`

`dis.disco(code, lasti=-1, *, file=None)`

Désassemble un code objet, en indiquant la dernière instruction si `lasti` est fournie. La sortie est répartie sur les colonnes suivantes :

1. le numéro de ligne, pour la première instruction de chaque ligne
2. l'instruction en cours, indiquée par `-->`,
3. une instruction libellée, indiquée par `> >`,
4. l'adresse de l'instruction,
5. le nom de le code d'opération,
6. paramètres de l'opération, et
7. interprétation des paramètres entre parenthèses.

L'interprétation du paramètre reconnaît les noms des variables locales et globales, des valeurs constantes, des branchements cibles, et des opérateurs de comparaison.

Le désassemblage est envoyé sous forme de texte à l'argument du fichier *file* si il est fourni, et à `sys.stdout` sinon.

Modifié dans la version 3.4 : Ajout du paramètre *file*.

`dis.get_instructions(x, *, first_line=None)`

Retourne un itérateur sur les instructions dans la fonction fournie, la méthode, les chaînes de caractères du code source ou objet.

Cet itérateur génère une série de n-uplets de *Instruction* qui donnent les détails de chacune des opérations dans le code fourni.

Si *first_line* ne vaut pas `None`, elle indique le nombre de la ligne qui doit être considérée comme première ligne source dans le code désassemblé. Autrement, les informations sur la ligne source sont prises directement à partir de la classe du code désassemblé.

Nouveau dans la version 3.4.

`dis.findlinestarts(code)`

This generator function uses the `co_firstlineno` and `co_lnotab` attributes of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as (`offset`, `lineno`) pairs. See [Objects/lnotab_notes.txt](#) for the `co_lnotab` format and how to decode it.

Modifié dans la version 3.6 : Les numéros de lignes peuvent être décroissants. Avant, ils étaient toujours croissants.

`dis.findlabels(code)`

Detect all offsets in the code object *code* which are jump targets, and return a list of these offsets.

`dis.stack_effect(opcode[, oparg])`

Compute the stack effect of *opcode* with argument *oparg*.

Nouveau dans la version 3.4.

32.12.3 Les instructions du code intermédiaire en Python

La fonction `get_instructions()` et la méthode *Bytecode* fournit des détails sur le code intermédiaire des instructions comme *Instruction* instances :

class `dis.Instruction`

Détails sur le code intermédiaire de l'opération

opcode

code numérique pour l'opération, correspondant aux valeurs de l'*opcode* ci-dessous et les valeurs du code intermédiaire dans la *Opcode collections*.

opname

nom lisible/compréhensible de l'opération

arg

le cas échéant, argument numérique de l'opération sinon `None`

argval

resolved arg value (if known), otherwise same as arg

argrepr

human readable description of operation argument

offset

start index of operation within bytecode sequence

starts_line

line started by this opcode (if any), otherwise `None`

is_jump_target

`True` if other code jumps to here, otherwise `False`

Nouveau dans la version 3.4.

The Python compiler currently generates the following bytecode instructions.

General instructions

NOP

Do nothing code. Used as a placeholder by the bytecode optimizer.

POP_TOP

Removes the top-of-stack (TOS) item.

ROT_TWO

Swaps the two top-most stack items.

ROT_THREE

Lifts second and third stack item one position up, moves top down to position three.

DUP_TOP

Duplicates the reference on top of the stack.

Nouveau dans la version 3.2.

DUP_TOP_TWO

Duplicates the two references on top of the stack, leaving them in the same order.

Nouveau dans la version 3.2.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_POSITIVE

Implements $TOS = +TOS$.

UNARY_NEGATIVE

Implements $TOS = -TOS$.

UNARY_NOT

Implements $TOS = \text{not } TOS$.

UNARY_INVERT

Implements $TOS = \sim TOS$.

GET_ITER

Implements $TOS = \text{iter}(TOS)$.

GET_YIELD_FROM_ITER

If TOS is a *generator iterator* or *coroutine* object it is left as is. Otherwise, implements $TOS = \text{iter}(TOS)$.

Nouveau dans la version 3.5.

Binary operations

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

BINARY_POWER

Implements $TOS = TOS1 ** TOS$.

BINARY_MULTIPLY

Implements $TOS = TOS1 * TOS$.

BINARY_MATRIX_MULTIPLY

Implements $TOS = TOS1 @ TOS$.

Nouveau dans la version 3.5.

BINARY_FLOOR_DIVIDE

Implements $TOS = TOS1 // TOS$.

BINARY_TRUE_DIVIDE

Implements $TOS = TOS1 / TOS$.

BINARY_MODULO

Implements $TOS = TOS1 \% TOS$.

BINARY_ADD

Implements $TOS = TOS1 + TOS$.

BINARY_SUBTRACT

Implements $TOS = TOS1 - TOS$.

BINARY_SUBSCR

Implements $TOS = TOS1[TOS]$.

BINARY_LSHIFT

Implements $TOS = TOS1 << TOS$.

BINARY_RSHIFT

Implements $TOS = TOS1 >> TOS$.

BINARY_AND

Implements $TOS = TOS1 \& TOS$.

BINARY_XOR

Implements $TOS = TOS1 \wedge TOS$.

BINARY_OR

Implements $TOS = TOS1 | TOS$.

In-place operations

In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

INPLACE_POWER

Implements in-place $TOS = TOS1 ** TOS$.

INPLACE_MULTIPLY

Implements in-place $TOS = TOS1 * TOS$.

INPLACE_MATRIX_MULTIPLY

Implements in-place $TOS = TOS1 @ TOS$.

Nouveau dans la version 3.5.

INPLACE_FLOOR_DIVIDE

Implements in-place $TOS = TOS1 // TOS$.

INPLACE_TRUE_DIVIDE

Implements in-place $TOS = TOS1 / TOS$.

INPLACE_MODULO

Implements in-place $TOS = TOS1 \% TOS$.

INPLACE_ADD

Implements in-place $TOS = TOS1 + TOS$.

INPLACE_SUBTRACT

Implements in-place $TOS = TOS1 - TOS$.

INPLACE_LSHIFT

Implements in-place $TOS = TOS1 << TOS$.

INPLACE_RSHIFT

Implements in-place `TOS = TOS1 >> TOS`.

INPLACE_AND

Implements in-place `TOS = TOS1 & TOS`.

INPLACE_XOR

Implements in-place `TOS = TOS1 ^ TOS`.

INPLACE_OR

Implements in-place `TOS = TOS1 | TOS`.

STORE_SUBSCR

Implements `TOS1[TOS] = TOS2`.

DELETE_SUBSCR

Implements `del TOS1[TOS]`.

Coroutine opcodes

GET_AWAITABLE

Implements `TOS = get_awaitable(TOS)`, where `get_awaitable(o)` returns `o` if `o` is a coroutine object or a generator object with the `CO_ITERABLE_COROUTINE` flag, or resolves `o.__await__`.

Nouveau dans la version 3.5.

GET_AITER

Implements `TOS = get_awaitable(TOS.__aiter__())`. See `GET_AWAITABLE` for details about `get_awaitable`.

Nouveau dans la version 3.5.

GET_ANEXT

Implements `PUSH(get_awaitable(TOS.__anext__()))`. See `GET_AWAITABLE` for details about `get_awaitable`.

Nouveau dans la version 3.5.

BEFORE_ASYNC_WITH

Resolves `__aenter__` and `__aexit__` from the object on top of the stack. Pushes `__aexit__` and result of `__aenter__()` to the stack.

Nouveau dans la version 3.5.

SETUP_ASYNC_WITH

Creates a new frame object.

Nouveau dans la version 3.5.

Miscellaneous opcodes

PRINT_EXPR

Implements the expression statement for the interactive mode. `TOS` is removed from the stack and printed. In non-interactive mode, an expression statement is terminated with `POP_TOP`.

BREAK_LOOP

Terminates a loop due to a `break` statement.

CONTINUE_LOOP (*target*)

Continues a loop due to a `continue` statement. *target* is the address to jump to (which should be a `FOR_ITER` instruction).

SET_ADD (*i*)

Calls `set.add(TOS1[-i], TOS)`. Used to implement set comprehensions.

LIST_APPEND (*i*)

Calls `list.append(TOS[-i], TOS)`. Used to implement list comprehensions.

MAP_ADD (*i*)

Calls `dict.setdefault(TOS1[-i], TOS, TOS1)`. Used to implement dict comprehensions.
Nouveau dans la version 3.1.

For all of the [SET_ADD](#), [LIST_APPEND](#) and [MAP_ADD](#) instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

RETURN_VALUE

Returns with TOS to the caller of the function.

YIELD_VALUE

Pops TOS and yields it from a *generator*.

YIELD_FROM

Pops TOS and delegates to it as a subiterator from a *generator*.

Nouveau dans la version 3.3.

SETUP_ANNOTATIONS

Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty dict. This opcode is only emitted if a class or module body contains *variable annotations* statically.

Nouveau dans la version 3.6.

IMPORT_STAR

Loads all symbols not starting with `'_'` directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements `from module import *`.

POP_BLOCK

Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

POP_EXCEPT

Removes one block from the block stack. The popped block must be an exception handler block, as implicitly created when entering an except handler. In addition to popping extraneous values from the frame stack, the last three popped values are used to restore the exception state.

END_FINALLY

Terminates a *finally* clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

LOAD_BUILD_CLASS

Pushes `builtins.__build_class__()` onto the stack. It is later called by [CALL_FUNCTION](#) to construct a class.

SETUP_WITH (*delta*)

This opcode performs several operations before a with block starts. First, it loads `__exit__()` from the context manager and pushes it onto the stack for later use by [WITH_CLEANUP](#). Then, `__enter__()` is called, and a finally block pointing to *delta* is pushed. Finally, the result of calling the enter method is pushed onto the stack. The next opcode will either ignore it ([POP_TOP](#)), or store it in (a) variable(s) ([STORE_FAST](#), [STORE_NAME](#), or [UNPACK_SEQUENCE](#)).

Nouveau dans la version 3.2.

WITH_CLEANUP_START

Cleans up the stack when a with statement block exits. TOS is the context manager's `__exit__()` bound method. Below TOS are 1–3 values indicating how/why the finally clause was entered :

- `SECOND = None`
- `(SECOND, THIRD) = (WHY_{RETURN, CONTINUE}), retval`
- `SECOND = WHY_*` ; no `retval` below it
- `(SECOND, THIRD, FOURTH) = exc_info()`

In the last case, `TOS(SECOND, THIRD, FOURTH)` is called, otherwise `TOS(None, None, None)`. Pushes `SECOND` and result of the call to the stack.

WITH_CLEANUP_FINISH

Pops exception type and result of “exit” function call from the stack.

If the stack represents an exception, *and* the function call returns a “true” value, this information is « zapped » and replaced with a single `WHY_SILENCED` to prevent `END_FINALLY` from re-raising the exception. (But non-local gotos will still be resumed.)

All of the following opcodes use their arguments.

STORE_NAME (*namei*)

Implements `name = TOS`. *namei* is the index of *name* in the attribute `co_names` of the code object. The compiler tries to use `STORE_FAST` or `STORE_GLOBAL` if possible.

DELETE_NAME (*namei*)

Implements `del name`, where *namei* is the index into `co_names` attribute of the code object.

UNPACK_SEQUENCE (*count*)

Unpacks TOS into *count* individual values, which are put onto the stack right-to-left.

UNPACK_EX (*counts*)

Implements assignment with a starred target : Unpacks an iterable in TOS into individual values, where the total number of values can be smaller than the number of items in the iterable : one of the new values will be a list of all leftover items.

The low byte of *counts* is the number of values before the list value, the high byte of *counts* the number of values after it. The resulting values are put onto the stack right-to-left.

STORE_ATTR (*namei*)

Implements `TOS.name = TOS1`, where *namei* is the index of *name* in `co_names`.

DELETE_ATTR (*namei*)

Implements `del TOS.name`, using *namei* as index into `co_names`.

STORE_GLOBAL (*namei*)

Works as `STORE_NAME`, but stores the name as a global.

DELETE_GLOBAL (*namei*)

Works as `DELETE_NAME`, but deletes a global name.

LOAD_CONST (*consti*)

Pushes `co_consts[consti]` onto the stack.

LOAD_NAME (*namei*)

Pushes the value associated with `co_names[namei]` onto the stack.

BUILD_TUPLE (*count*)

Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack.

BUILD_LIST (*count*)

Works as `BUILD_TUPLE`, but creates a list.

BUILD_SET (*count*)

Works as `BUILD_TUPLE`, but creates a set.

BUILD_MAP (*count*)

Pushes a new dictionary object onto the stack. Pops $2 * \text{count}$ items so that the dictionary holds *count* entries : `{..., TOS3: TOS2, TOS1: TOS}`.

Modifié dans la version 3.5 : The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold *count* items.

BUILD_CONST_KEY_MAP (*count*)

The version of `BUILD_MAP` specialized for constant keys. *count* values are consumed from the stack. The top element on the stack contains a tuple of keys.

Nouveau dans la version 3.6.

BUILD_STRING (*count*)

Concatenates *count* strings from the stack and pushes the resulting string onto the stack.

Nouveau dans la version 3.6.

BUILD_TUPLE_UNPACK (*count*)

Pops *count* iterables from the stack, joins them in a single tuple, and pushes the result. Implements iterable unpacking in tuple displays (**x*, **y*, **z*).

Nouveau dans la version 3.5.

BUILD_TUPLE_UNPACK_WITH_CALL (*count*)

This is similar to [BUILD_TUPLE_UNPACK](#), but is used for *f* (**x*, **y*, **z*) call syntax. The stack item at position *count* + 1 should be the corresponding callable *f*.

Nouveau dans la version 3.6.

BUILD_LIST_UNPACK (*count*)

This is similar to [BUILD_TUPLE_UNPACK](#), but pushes a list instead of tuple. Implements iterable unpacking in list displays [**x*, **y*, **z*].

Nouveau dans la version 3.5.

BUILD_SET_UNPACK (*count*)

This is similar to [BUILD_TUPLE_UNPACK](#), but pushes a set instead of tuple. Implements iterable unpacking in set displays {**x*, **y*, **z*}.

Nouveau dans la version 3.5.

BUILD_MAP_UNPACK (*count*)

Pops *count* mappings from the stack, merges them into a single dictionary, and pushes the result. Implements dictionary unpacking in dictionary displays {***x*, ***y*, ***z*}.

Nouveau dans la version 3.5.

BUILD_MAP_UNPACK_WITH_CALL (*count*)

This is similar to [BUILD_MAP_UNPACK](#), but is used for *f* (***x*, ***y*, ***z*) call syntax. The stack item at position *count* + 2 should be the corresponding callable *f*.

Nouveau dans la version 3.5.

Modifié dans la version 3.6 : The position of the callable is determined by adding 2 to the opcode argument instead of encoding it in the second byte of the argument.

LOAD_ATTR (*namei*)

Replaces TOS with `getattr(TOS, co_names[namei])`.

COMPARE_OP (*opname*)

Performs a Boolean operation. The operation name can be found in `cmp_op[opname]`.

IMPORT_NAME (*namei*)

Imports the module `co_names[namei]`. TOS and TOS1 are popped and provide the *fromlist* and *level* arguments of `__import__()`. The module object is pushed onto the stack. The current namespace is not affected : for a proper import statement, a subsequent [STORE_FAST](#) instruction modifies the namespace.

IMPORT_FROM (*namei*)

Loads the attribute `co_names[namei]` from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a [STORE_FAST](#) instruction.

JUMP_FORWARD (*delta*)

Increments bytecode counter by *delta*.

POP_JUMP_IF_TRUE (*target*)

If TOS is true, sets the bytecode counter to *target*. TOS is popped.

Nouveau dans la version 3.1.

POP_JUMP_IF_FALSE (*target*)

If TOS is false, sets the bytecode counter to *target*. TOS is popped.

Nouveau dans la version 3.1.

JUMP_IF_TRUE_OR_POP (*target*)

If TOS is true, sets the bytecode counter to *target* and leaves TOS on the stack. Otherwise (TOS is false), TOS is popped.

Nouveau dans la version 3.1.

JUMP_IF_FALSE_OR_POP (*target*)

If TOS is false, sets the bytecode counter to *target* and leaves TOS on the stack. Otherwise (TOS is true), TOS is popped.

Nouveau dans la version 3.1.

JUMP_ABSOLUTE (*target*)

Set bytecode counter to *target*.

FOR_ITER (*delta*)

TOS is an *iterator*. Call its `__next__()` method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the byte code counter is incremented by *delta*.

LOAD_GLOBAL (*namei*)

Loads the global named `co_names[namei]` onto the stack.

SETUP_LOOP (*delta*)

Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of *delta* bytes.

SETUP_EXCEPT (*delta*)

Pushes a try block from a try-except clause onto the block stack. *delta* points to the first except block.

SETUP_FINALLY (*delta*)

Pushes a try block from a try-except clause onto the block stack. *delta* points to the finally block.

LOAD_FAST (*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack.

STORE_FAST (*var_num*)

Stores TOS into the local `co_varnames[var_num]`.

DELETE_FAST (*var_num*)

Deletes local `co_varnames[var_num]`.

STORE_ANNOTATION (*namei*)

Stores TOS as `locals()['__annotations__'][co_names[namei]] = TOS`.

Nouveau dans la version 3.6.

LOAD_CLOSURE (*i*)

Pushes a reference to the cell contained in slot *i* of the cell and free variable storage. The name of the variable is `co_cellvars[i]` if *i* is less than the length of `co_cellvars`. Otherwise it is `co_freevars[i - len(co_cellvars)]`.

LOAD_DEREF (*i*)

Loads the cell contained in slot *i* of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

LOAD_CLASSDEREF (*i*)

Much like `LOAD_DEREF` but first checks the locals dictionary before consulting the cell. This is used for loading free variables in class bodies.

Nouveau dans la version 3.4.

STORE_DEREF (*i*)

Stores TOS into the cell contained in slot *i* of the cell and free variable storage.

DELETE_DEREF (*i*)

Empties the cell contained in slot *i* of the cell and free variable storage. Used by the `del` statement.

Nouveau dans la version 3.2.

RAISE_VARARGS (*argc*)

Raises an exception. *argc* indicates the number of arguments to the raise statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

CALL_FUNCTION (*argc*)

Calls a callable object with positional arguments. *argc* indicates the number of positional arguments. The top of the stack contains positional arguments, with the right-most argument on top. Below the arguments is a callable object to call. **CALL_FUNCTION** pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Modifié dans la version 3.6 : This opcode is used only for calls with positional arguments.

CALL_FUNCTION_KW (*argc*)

Calls a callable object with positional (if any) and keyword arguments. *argc* indicates the total number of positional and keyword arguments. The top element on the stack contains a tuple of keyword argument names. Below that are keyword arguments in the order corresponding to the tuple. Below that are positional arguments, with the right-most parameter on top. Below the arguments is a callable object to call. **CALL_FUNCTION_KW** pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Modifié dans la version 3.6 : Keyword arguments are packed in a tuple instead of a dictionary, *argc* indicates the total number of arguments.

CALL_FUNCTION_EX (*flags*)

Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Below that is an iterable object containing positional arguments and a callable object to call. **BUILD_MAP_UNPACK_WITH_CALL** and **BUILD_TUPLE_UNPACK_WITH_CALL** can be used for merging multiple mapping objects and iterables containing arguments. Before the callable is called, the mapping object and iterable object are each « unpacked » and their contents passed in as keyword and positional arguments respectively. **CALL_FUNCTION_EX** pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Nouveau dans la version 3.6.

MAKE_FUNCTION (*argc*)

Pushes a new function object on the stack. From bottom to top, the consumed stack must consist of values if the argument carries a specified flag value

- 0x01 a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- 0x02 a dictionary of keyword-only parameters' default values
- 0x04 an annotation dictionary
- 0x08 a tuple containing cells for free variables, making a closure
- the code associated with the function (at TOS1)
- the *qualified name* of the function (at TOS)

BUILD_SLICE (*argc*)

Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, `slice(TOS1, TOS)` is pushed; if it is 3, `slice(TOS2, TOS1, TOS)` is pushed. See the `slice()` built-in function for more information.

EXTENDED_ARG (*ext*)

Prefixes any opcode which has an argument too big to fit into the default two bytes. *ext* holds two additional bytes which, taken together with the subsequent opcode's argument, comprise a four-byte argument, *ext* being the two most-significant bytes.

FORMAT_VALUE (*flags*)

Used for implementing formatted literal strings (f-strings). Pops an optional *fmt_spec* from the stack, then a required *value*. *flags* is interpreted as follows :

- (flags & 0x03) == 0x00 : *value* is formatted as-is.
- (flags & 0x03) == 0x01 : call *str()* on *value* before formatting it.
- (flags & 0x03) == 0x02 : call *repr()* on *value* before formatting it.
- (flags & 0x03) == 0x03 : call *ascii()* on *value* before formatting it.
- (flags & 0x04) == 0x04 : pop *fmt_spec* from the stack and use it, else use an empty *fmt_spec*.

Formatting is performed using `PyObject_Format()`. The result is pushed on the stack.

Nouveau dans la version 3.6.

HAVE_ARGUMENT

This is not really an opcode. It identifies the dividing line between opcodes which don't use their argument and those that do (< HAVE_ARGUMENT and >= HAVE_ARGUMENT, respectively).

Modifié dans la version 3.6 : Now every instruction has an argument, but opcodes < HAVE_ARGUMENT ignore it. Before, only opcodes >= HAVE_ARGUMENT had an argument.

32.12.4 Opcode collections

These collections are provided for automatic introspection of bytecode instructions :

dis.opname

Sequence of operation names, indexable using the bytecode.

dis.opmap

Dictionary mapping operation names to bytecodes.

dis.cmp_op

Sequence of all compare operation names.

dis.hasconst

Sequence of bytecodes that access a constant.

dis.hasfree

Sequence of bytecodes that access a free variable (note that “free” in this context refers to names in the current scope that are referenced by inner scopes or names in outer scopes that are referenced from this scope. It does *not* include references to global or builtin scopes).

dis.hasname

Sequence of bytecodes that access an attribute by name.

dis.hasjrel

Sequence of bytecodes that have a relative jump target.

dis.hasjabs

Sequence of bytecodes that have an absolute jump target.

dis.haslocal

Sequence of bytecodes that access a local variable.

dis.hascompare

Sequence of bytecodes of Boolean operations.

32.13 `pickletools` — Tools for pickle developers

Source code : [Lib/pickletools.py](#)

This module contains various constants relating to the intimate details of the `pickle` module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the `pickle`; ordinary users of the `pickle` module probably won't find the `pickletools` module relevant.

32.13.1 Utilisation de la ligne de commande

Nouveau dans la version 3.2.

When invoked from the command line, `python -m pickletools` will disassemble the contents of one or more pickle files. Note that if you want to see the Python object stored in the pickle rather than the details of pickle format, you may want to use `-m pickle` instead. However, when the pickle file that you want to examine comes from an untrusted source, `-m pickletools` is a safer option because it does not execute pickle bytecode.

For example, with a tuple `(1, 2)` pickled in file `x.pickle`:

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
0: \x80 PROTO      3
2: K    BININT1    1
4: K    BININT1    2
6: \x86 TUPLE2
7: q    BINPUT     0
9: .    STOP
highest protocol among opcodes = 2
```

Options de la ligne de commande

- a, --annotate**
Annotate each line with a short opcode description.
- o, --output=<file>**
Name of a file where the output should be written.
- l, --indentlevel=<num>**
The number of blanks by which to indent a new MARK level.
- m, --memo**
When multiple objects are disassembled, preserve memo between disassemblies.
- p, --preamble=<preamble>**
When more than one pickle file are specified, print given preamble before each disassembly.

32.13.2 Programmatic Interface

`pickletools.dis` (*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)

Outputs a symbolic disassembly of the pickle to the file-like object *out*, defaulting to `sys.stdout`. *pickle* can be a string or a file-like object. *memo* can be a Python dictionary that will be used as the pickle's memo; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by `MARK` opcodes in the stream, are indented by *indentlevel* spaces. If a nonzero value is given to *annotate*, each opcode in the output is annotated with a short description. The value of *annotate* is used as a hint for the column where annotation should start.

Nouveau dans la version 3.2 : The *annotate* argument.

`pickletools.genops` (*pickle*)

Provides an *iterator* over all of the opcodes in a pickle, returning a sequence of (*opcode*, *arg*, *pos*) triples. *opcode* is an instance of an `OpcodeInfo` class; *arg* is the decoded value, as a Python object, of the opcode's argument; *pos* is the position at which this opcode is located. *pickle* can be a string or a file-like object.

`pickletools.optimize` (*picklestring*)

Returns a new equivalent pickle string after eliminating unused `PUT` opcodes. The optimized pickle is shorter, takes less transmission time, requires less storage space, and unpickles more efficiently.

Les modules documentés dans ce chapitre fournissent différents services disponibles dans toutes les versions de Python. En voici un aperçu :

33.1 `formatter` — Generic output formatting

Obsolète depuis la version 3.4 : Due to lack of usage, the `formatter` module has been deprecated.

This module supports two interface definitions, each with multiple implementations : The *formatter* interface, and the *writer* interface which is required by the *formatter* interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored ; writers need not be able to handle relative changes nor any sort of « change back » operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.

33.1.1 The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level :

`formatter.AS_IS`

Value which can be used in the font specification passed to the `push_font()` method described below, or as the new value to any other `push_property()` method. Pushing the `AS_IS` value allows the corresponding `pop_property()` method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects :

`formatter.writer`

The writer instance with which the formatter interacts.

`formatter.end_paragraph(blanklines)`

Close any open paragraphs and insert at least *blanklines* before the next paragraph.

`formatter.add_line_break()`

Add a hard line break if one does not already exist. This does not break the logical paragraph.

`formatter.add_hor_rule(*args, **kw)`

Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer's `send_line_break()` method.

`formatter.add_flow_data(data)`

Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to `add_flow_data()` is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

`formatter.add_literal_data(data)`

Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of *data*.

`formatter.add_label_data(format, counter)`

Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the *format* value is a string, it is interpreted as a format specification for *counter*, which should be an integer. The result of this formatting becomes the value of the label; if *format* is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's `send_label_data()` method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character '1' represents the counter value formatter as an Arabic number, the characters 'A' and 'a' represent alphabetic representations of the counter value in upper and lower case, respectively, and 'I' and 'i' represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

`formatter.flush_softspace()`

Send any pending whitespace buffered from a previous call to `add_flow_data()` to the associated writer object. This should be called before any direct manipulation of the writer object.

`formatter.push_alignment(align)`

Push a new alignment setting onto the alignment stack. This may be `AS_IS` if no change is desired. If the alignment value is changed from the previous setting, the writer's `new_alignment()` method is called with the *align* value.

`formatter.pop_alignment()`

Restore the previous alignment.

`formatter.push_font((size, italic, bold, teletype))`

Change some or all font properties of the writer object. Properties which are not set to `AS_IS` are set to the values passed in while others are maintained at their current settings. The writer's `new_font()` method is called with the fully resolved font specification.

`formatter.pop_font()`

Restore the previous font.

`formatter.push_margin(margin)`

Increase the number of left margin indentations by one, associating the logical tag *margin* with the new indentation. The initial margin level is 0. Changed values of the logical tag must be true values; false values other than `AS_IS` are not sufficient to change the margin.

`formatter.pop_margin()`

Restore the previous margin.

`formatter.push_style(*styles)`

Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including `AS_IS` values, is passed to the writer's `new_styles()` method.

`formatter.pop_style(n=1)`

Pop the last *n* style specifications passed to `push_style()`. A tuple representing the revised stack, including `AS_IS` values, is passed to the writer's `new_styles()` method.

`formatter.set_spacing(spacing)`

Set the spacing style for the writer.

`formatter.assert_line_data(flag=1)`

Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional *flag* argument can be set to false if the writer manipulations produced a hard line break at the end of the output.

33.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

class `formatter.NullFormatter(writer=None)`

A formatter which does nothing. If *writer* is omitted, a `NullWriter` instance is created. No methods of the writer are called by `NullFormatter` instances. Implementations should inherit from this class if implementing a writer interface but don't need to inherit any implementation.

class `formatter.AbstractFormatter(writer)`

The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.

33.1.3 The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the `AbstractFormatter` class as a formatter, the writer must typically be provided by the application.

`writer.flush()`

Flush any buffered output or device control events.

`writer.new_alignment(align)`

Set the alignment style. The *align* value can be any object, but by convention is a string or `None`, where `None` indicates that the writer's « preferred » alignment should be used. Conventional *align* values are 'left', 'center', 'right', and 'justify'.

`writer.new_font(font)`

Set the font style. The value of *font* will be `None`, indicating that the device's default font should be used, or a tuple of the form (*size*, *italic*, *bold*, *teletype*). *Size* will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The *italic*, *bold*, and *teletype* values are Boolean values specifying which of those font attributes should be used.

`writer.new_margin(margin, level)`

Set the margin level to the integer *level* and the logical tag to *margin*. Interpretation of the logical tag is at the writer's discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of *level*.

`writer.new_spacing(spacing)`

Set the spacing style to *spacing*.

`writer.new_styles(styles)`

Set additional styles. The *styles* value is a tuple of arbitrary values; the value `AS_IS` should be ignored. The *styles* tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

`writer.send_line_break()`

Break the current line.

`writer.send_paragraph(blankline)`

Produce a paragraph separation of at least *blankline* blank lines, or the equivalent. The *blankline* value will be an integer. Note that the implementation will receive a call to `send_line_break()` before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

`writer.send_hor_rule(*args, **kw)`

Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via `send_line_break()`.

`writer.send_flow_data(data)`

Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

`writer.send_literal_data(data)`

Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the `send_formatted_data()` interface.

`writer.send_label_data(data)`

Set *data* to the left of the current left margin, if possible. The value of *data* is not restricted; treatment of non-string

values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.

33.1.4 Writer Implementations

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the *NullWriter* class.

class `formatter.NullWriter`

A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class `formatter.AbstractWriter`

A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class `formatter.DumbWriter` (*file=None, maxcol=72*)

Simple writer class which writes output on the *file object* passed in as *file* or, if *file* is omitted, on standard output. The output is simply word-wrapped to the number of columns specified by *maxcol*. This class is suitable for reflowing a sequence of paragraphs.

Services spécifiques à MS Windows

Ce chapitre documente les modules qui ne sont disponibles que sous MS Windows.

34.1 `msilib` — Read and write Microsoft Installer files

Source code : [Lib/msilib/__init__.py](#)

The `msilib` supports the creation of Microsoft Installer (`.msi`) files. Because these files often contain an embedded « cabinet » file (`.cab`), it also exposes an API to create CAB files. Support for reading `.cab` files is currently not implemented; read support for the `.msi` database is possible.

This package aims to provide complete access to all tables in an `.msi` file, therefore, it is a fairly low-level API. Two primary applications of this package are the `distutils` command `bdist_msi`, and the creation of Python installer package itself (although that currently uses a different version of `msilib`).

The package contents can be roughly split into four parts : low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

`msilib.FCICreate` (*cabname*, *files*)

Create a new CAB file named *cabname*. *files* must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

Callbacks to Python for the various steps of MSI creation are currently not exposed.

`msilib.UuidCreate` ()

Return the string representation of a new unique identifier. This wraps the Windows API functions `UuidCreate` () and `UuidToString` ().

`msilib.OpenDatabase` (*path*, *persist*)

Return a new database object by calling `MsiOpenDatabase`. *path* is the file name of the MSI file; *persist* can be one of the constants `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`, `MSIDBOPEN_DIRECT`,

MSIDBOPEN_READONLY, or MSIDBOPEN_TRANSACT, and may include the flag MSIDBOPEN_PATCHFILE. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

`msilib.CreateRecord(count)`

Return a new record object by calling `MSICreateRecord()`. *count* is the number of fields of the record.

`msilib.init_database(name, schema, ProductName, ProductCode, ProductVersion, Manufacturer)`

Create and return a new database *name*, initialize it with *schema*, and set the properties *ProductName*, *ProductCode*, *ProductVersion*, and *Manufacturer*.

schema must be a module object containing `tables` and `_Validation_records` attributes; typically, `msilib.schema` should be used.

The database will contain just the schema and the validation records when this function returns.

`msilib.add_data(database, table, records)`

Add all *records* to the table named *table* in *database*.

The *table* argument must be one of the predefined tables in the MSI schema, e.g. 'Feature', 'File', 'Component', 'Dialog', 'Control', etc.

records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, `None` can be passed.

Field values can be ints, strings, or instances of the `Binary` class.

class `msilib.Binary(filename)`

Represents entries in the Binary table; inserting such an object using `add_data()` reads the file named *filename* into the table.

`msilib.add_tables(database, module)`

Add all table content from *module* to *database*. *module* must contain an attribute *tables* listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

`msilib.add_stream(database, name, path)`

Add the file *path* into the `_Stream` table of *database*, with the stream name *name*.

`msilib.gen_uuid()`

Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in upper-case).

Voir aussi :

`FCICreate UuidCreate UuidToString`

34.1.1 Database Objects

`Database.OpenView(sql)`

Return a view object, by calling `MSIDatabaseOpenView()`. *sql* is the SQL statement to execute.

`Database.Commit()`

Commit the changes pending in the current transaction, by calling `MSIDatabaseCommit()`.

`Database.GetSummaryInformation(count)`

Return a new summary information object, by calling `MsiGetSummaryInformation()`. *count* is the maximum number of updated values.

Voir aussi :

`MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation`

34.1.2 View Objects

`View.Execute(params)`

Execute the SQL query of the view, through `MsiViewExecute()`. If *params* is not `None`, it is a record describing actual values of the parameter tokens in the query.

`View.GetColumnInfo(kind)`

Return a record describing the columns of the view, through calling `MsiViewGetColumnInfo()`. *kind* can be either `MSICOLINFO_NAMES` or `MSICOLINFO_TYPES`.

`View.Fetch()`

Return a result record of the query, through calling `MsiViewFetch()`.

`View.Modify(kind, data)`

Modify the view, by calling `MsiViewModify()`. *kind* can be one of `MSIMODIFY_SEEK`, `MSIMODIFY_REFRESH`, `MSIMODIFY_INSERT`, `MSIMODIFY_UPDATE`, `MSIMODIFY_ASSIGN`, `MSIMODIFY_REPLACE`, `MSIMODIFY_MERGE`, `MSIMODIFY_DELETE`, `MSIMODIFY_INSERT_TEMPORARY`, `MSIMODIFY_VALIDATE`, `MSIMODIFY_VALIDATE_NEW`, `MSIMODIFY_VALIDATE_FIELD`, or `MSIMODIFY_VALIDATE_DELETE`.

data must be a record describing the new data.

`View.Close()`

Close the view, through `MsiViewClose()`.

Voir aussi :

[MsiViewExecute](#) [MsiViewGetColumnInfo](#) [MsiViewFetch](#) [MsiViewModify](#) [MsiViewClose](#)

34.1.3 Summary Information Objects

`SummaryInformation.GetProperty(field)`

Return a property of the summary, through `MsiSummaryInfoGetProperty()`. *field* is the name of the property, and can be one of the constants `PID_CODEPAGE`, `PID_TITLE`, `PID_SUBJECT`, `PID_AUTHOR`, `PID_KEYWORDS`, `PID_COMMENTS`, `PID_TEMPLATE`, `PID_LASTAUTHOR`, `PID_REVNUMBER`, `PID_LASTPRINTED`, `PID_CREATE_DTM`, `PID_LASTSAVE_DTM`, `PID_PAGECOUNT`, `PID_WORDCOUNT`, `PID_CHARCOUNT`, `PID_APPNAME`, or `PID_SECURITY`.

`SummaryInformation.GetPropertyCount()`

Return the number of summary properties, through `MsiSummaryInfoGetPropertyCount()`.

`SummaryInformation.SetProperty(field, value)`

Set a property through `MsiSummaryInfoSetProperty()`. *field* can have the same values as in [GetProperty\(\)](#), *value* is the new value of the property. Possible value types are integer and string.

`SummaryInformation.Persist()`

Write the modified properties to the summary information stream, using `MsiSummaryInfoPersist()`.

Voir aussi :

[MsiSummaryInfoGetProperty](#) [MsiSummaryInfoGetPropertyCount](#) [MsiSummaryInfoSetProperty](#) [MsiSummaryInfoPersist](#)

34.1.4 Record Objects

`Record.GetFieldCount()`

Return the number of fields of the record, through `MsiRecordGetFieldCount()`.

`Record.GetInteger(field)`

Return the value of *field* as an integer where possible. *field* must be an integer.

`Record.GetString(field)`

Return the value of *field* as a string where possible. *field* must be an integer.

`Record.SetString(field, value)`

Set *field* to *value* through `MsiRecordSetString()`. *field* must be an integer; *value* a string.

`Record.SetStream(field, value)`

Set *field* to the contents of the file named *value*, through `MsiRecordSetStream()`. *field* must be an integer; *value* a string.

`Record.SetInteger(field, value)`

Set *field* to *value* through `MsiRecordSetInteger()`. Both *field* and *value* must be an integer.

`Record.ClearData()`

Set all fields of the record to 0, through `MsiRecordClearData()`.

Voir aussi :

[MsiRecordGetFieldCount](#) [MsiRecordSetString](#) [MsiRecordSetStream](#) [MsiRecordSetInteger](#) [MsiRecordClearData](#)

34.1.5 Errors

All wrappers around MSI functions raise `MSIError`; the string inside the exception will contain more detail.

34.1.6 CAB Objects

class `msilib.CAB(name)`

The class `CAB` represents a CAB file. During MSI construction, files will be added simultaneously to the `Files` table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

name is the name of the CAB file in the MSI file.

append (*full, file, logical*)

Add the file with the pathname *full* to the CAB file, under the name *logical*. If there is already a file named *logical*, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

commit (*database*)

Generate a CAB file, add it as a stream to the MSI file, put it into the `Media` table, and remove the generated file from the disk.

34.1.7 Directory Objects

class `msilib.Directory` (*database, cab, basedir, physical, logical, default[, componentflags]*)

Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through `start_component()`, or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be `None`), the path to the physical directory, and a logical directory name. *default* specifies the DefaultDir slot in the directory table. *componentflags* specifies the default flags that new components get.

start_component (*component=None, feature=None, flags=None, keyfile=None, uuid=None*)

Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no *feature* is given, the current feature is used. If no *flags* are given, the directory's default flags are used. If no *keyfile* is given, the KeyPath is left null in the Component table.

add_file (*file, src=None, version=None, language=None*)

Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the *src* file is specified, it is interpreted relative to the current directory. Optionally, a *version* and a *language* can be specified for the entry in the File table.

glob (*pattern, exclude=None*)

Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the *exclude* list.

remove_pyc ()

Remove `.pyc` files on uninstall.

Voir aussi :

[Directory Table](#) [File Table](#) [Component Table](#) [FeatureComponents Table](#)

34.1.8 Features

class `msilib.Feature` (*db, id, title, desc, display, level=1, parent=None, directory=None, attributes=0*)

Add a new record to the Feature table, using the values *id*, *parent.id*, *title*, *desc*, *display*, *level*, *directory*, and *attributes*. The resulting feature object can be passed to the `start_component()` method of `Directory`.

set_current ()

Make this feature the current feature of `msilib`. New components are automatically added to the default feature, unless a feature is explicitly specified.

Voir aussi :

[Feature Table](#)

34.1.9 GUI classes

`msilib` provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use `bdist_msi` to create MSI files with a user-interface for installing Python packages.

class `msilib.Control` (*dlg, name*)

Base class of the dialog controls. *dlg* is the dialog object the control belongs to, and *name* is the control's name.

event (*event, argument, condition=1, ordering=None*)

Make an entry into the `ControlEvent` table for this control.

mapping (*event, attribute*)

Make an entry into the `EventMapping` table for this control.

condition (*action, condition*)

Make an entry into the `ControlCondition` table for this control.

class `msilib.RadioButtonGroup` (*dlg, name, property*)

Create a radio button control named *name*. *property* is the installer property that gets set when a radio button is selected.

add (*name, x, y, width, height, text, value=None*)

Add a radio button named *name* to the group, at the coordinates *x, y, width, height*, and with the label *text*. If *value* is `None`, it defaults to *name*.

class `msilib.Dialog` (*db, name, x, y, w, h, attr, title, first, default, cancel*)

Return a new `Dialog` object. An entry in the `Dialog` table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

control (*name, type, x, y, width, height, attributes, property, text, control_next, help*)

Return a new `Control` object. An entry in the `Control` table is made with the specified parameters.

This is a generic method ; for specific types, specialized methods are provided.

text (*name, x, y, width, height, attributes, text*)

Add and return a `Text` control.

bitmap (*name, x, y, width, height, text*)

Add and return a `Bitmap` control.

line (*name, x, y, width, height*)

Add and return a `Line` control.

pushbutton (*name, x, y, width, height, attributes, text, next_control*)

Add and return a `PushButton` control.

radiogroup (*name, x, y, width, height, attributes, property, text, next_control*)

Add and return a `RadioButtonGroup` control.

checkbox (*name, x, y, width, height, attributes, property, text, next_control*)

Add and return a `CheckBox` control.

Voir aussi :

[Dialog Table](#) [Control Table](#) [Control Types](#) [ControlCondition Table](#) [ControlEvents Table](#) [EventMapping Table](#) [RadioButton Table](#)

34.1.10 Precomputed tables

`msilib` provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

`msilib.schema`

This is the standard MSI schema for MSI 2.0, with the *tables* variable providing a list of table definitions, and *_Validation_records* providing the data for MSI validation.

`msilib.sequence`

This module contains table contents for the standard sequence tables : *AdminExecuteSequence*, *AdminUISequence*, *AdvtExecuteSequence*, *InstallExecuteSequence*, and *InstallUISequence*.

`msilib.text`

This module contains definitions for the *UIText* and *ActionText* tables, for the standard installer actions.

34.2 msvcrt — Useful routines from the MS VC++ runtime

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the `getpass` module uses this in the implementation of the `getpass()` function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used where ever possible.

Modifié dans la version 3.3 : Operations in this module now raise `OSError` where `IOError` was raised.

34.2.1 File Operations

`msvcrt.locking` (*fd*, *mode*, *nbytes*)

Lock part of a file based on file descriptor *fd* from the C runtime. Raises `OSError` on failure. The locked region of the file extends from the current file position for *nbytes* bytes, and may continue beyond the end of the file. *mode* must be one of the `LK_*` constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

`msvcrt.LK_LOCK`

`msvcrt.LK_RLCK`

Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, `OSError` is raised.

`msvcrt.LK_NBLCK`

`msvcrt.LK_NBLCK`

Locks the specified bytes. If the bytes cannot be locked, `OSError` is raised.

`msvcrt.LK_UNLCK`

Unlocks the specified bytes, which must have been previously locked.

`msvcrt.setmode` (*fd*, *flags*)

Set the line-end translation mode for the file descriptor *fd*. To set it to text mode, *flags* should be `os.O_TEXT`; for binary, it should be `os.O_BINARY`.

`msvcrt.open_osfhandle` (*handle*, *flags*)

Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bitwise OR of `os.O_APPEND`, `os.O_RDONLY`, and `os.O_TEXT`. The returned file descriptor may be used as a parameter to `os.fdopen()` to create a file object.

`msvcrt.get_osfhandle` (*fd*)

Return the file handle for the file descriptor *fd*. Raises `OSError` if *fd* is not recognized.

34.2.2 Console I/O

`msvcrt.kbhit()`

Return true if a keypress is waiting to be read.

`msvcrt.getch()`

Read a keypress and return the resulting character as a byte string. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for Enter to be pressed. If the pressed key was a special function key, this will return `'\000'` or `'\xe0'`; the next call will return the keycode. The Control-C keypress cannot be read with this function.

`msvcrt.getwch()`

Wide char variant of `getch()`, returning a Unicode value.

`msvcrt.getche()`

Similar to `getch()`, but the keypress will be echoed if it represents a printable character.

`msvcrt.getwche()`

Wide char variant of `getche()`, returning a Unicode value.

`msvcrt.putch(char)`

Print the byte string `char` to the console without buffering.

`msvcrt.putwch(unicode_char)`

Wide char variant of `putch()`, accepting a Unicode value.

`msvcrt.ungetch(char)`

Cause the byte string `char` to be « pushed back » into the console buffer; it will be the next character read by `getch()` or `getche()`.

`msvcrt.ungetwch(unicode_char)`

Wide char variant of `ungetch()`, accepting a Unicode value.

34.2.3 Other Functions

`msvcrt.heapmin()`

Force the `malloc()` heap to clean itself up and return unused blocks to the operating system. On failure, this raises `OSError`.

34.3 winreg — Windows registry access

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a *handle object* is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

Modifié dans la version 3.3 : Several functions in this module used to raise a `WindowsError`, which is now an alias of `OSError`.

34.3.1 Fonctions

This module offers the following functions :

`winreg.CloseKey` (*hkey*)

Closes a previously opened registry key. The *hkey* argument specifies a previously opened key.

Note : If *hkey* is not closed using this method (or via *hkey.Close()*), it is closed when the *hkey* object is destroyed by Python.

`winreg.ConnectRegistry` (*computer_name*, *key*)

Establishes a connection to a predefined registry handle on another computer, and returns a *handle object*.

computer_name is the name of the remote computer, of the form `r"\\computername"`. If `None`, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an *OSError* exception is raised.

Modifié dans la version 3.3 : See *above*.

`winreg.CreateKey` (*key*, *sub_key*)

Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an *OSError* exception is raised.

Modifié dans la version 3.3 : See *above*.

`winreg.CreateKeyEx` (*key*, *sub_key*, *reserved*=0, *access*=*KEY_WRITE*)

Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_WRITE*. See *Access Rights* for other allowed values.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an *OSError* exception is raised.

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : See *above*.

`winreg.DeleteKey` (*key*, *sub_key*)

Deletes the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an *OSError* exception is raised.

Modifié dans la version 3.3 : See *above*.

`winreg.DeleteKeyEx(key, sub_key, access=KEY_WOW64_64KEY, reserved=0)`

Deletes the specified key.

Note : The `DeleteKeyEx()` function is implemented with the `RegDeleteKeyEx` Windows API function, which is specific to 64-bit versions of Windows. See the [RegDeleteKeyEx documentation](#).

key is an already open key, or one of the predefined `HKEY_* constants`.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WOW64_64KEY`. See [Access Rights](#) for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an `OSError` exception is raised.

On unsupported Windows versions, `NotImplementedError` is raised.

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : See [above](#).

`winreg.DeleteValue(key, value)`

Removes a named value from a registry key.

key is an already open key, or one of the predefined `HKEY_* constants`.

value is a string that identifies the value to remove.

`winreg.EnumKey(key, index)`

Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or one of the predefined `HKEY_* constants`.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an `OSError` exception is raised, indicating, no more values are available.

Modifié dans la version 3.3 : See [above](#).

`winreg.EnumValue(key, index)`

Enumerates values of an open registry key, returning a tuple.

key is an already open key, or one of the predefined `HKEY_* constants`.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an `OSError` exception is raised, indicating no more values.

The result is a tuple of 3 items :

Index	Signification
0	A string that identifies the value name
1	An object that holds the value data, and whose type depends on the underlying registry type
2	An integer that identifies the type of the value data (see table in docs for SetValueEx())

Modifié dans la version 3.3 : See [above](#).

`winreg.ExpandEnvironmentStrings(str)`

Expands environment variable placeholders `%NAME%` in strings like `REG_EXPAND_SZ` :

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

`winreg.FlushKey(key)`

Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined *HKEY_* constants*.

It is not necessary to call *FlushKey()* to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike *CloseKey()*, the *FlushKey()* method returns only when all the data has been written to the registry. An application should only call *FlushKey()* if it requires absolute certainty that registry changes are on disk.

Note : If you don't know whether a *FlushKey()* call is required, it probably isn't.

`winreg.LoadKey(key, sub_key, file_name)`

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by *ConnectRegistry()* or one of the constants *HKEY_USERS* or *HKEY_LOCAL_MACHINE*.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the *SaveKey()* function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to *LoadKey()* fails if the calling process does not have the *SE_RESTORE_PRIVILEGE* privilege. Note that privileges are different from permissions – see the *RegLoadKey documentation* for more details.

If *key* is a handle returned by *ConnectRegistry()*, then the path specified in *file_name* is relative to the remote computer.

`winreg.OpenKey(key, sub_key, reserved=0, access=KEY_READ)`

`winreg.OpenKeyEx(key, sub_key, reserved=0, access=KEY_READ)`

Opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that identifies the sub_key to open.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_READ*. See *Access Rights* for other allowed values.

The result is a new handle to the specified key.

If the function fails, *OSError* is raised.

Modifié dans la version 3.2 : Allow the use of named arguments.

Modifié dans la version 3.3 : See *above*.

`winreg.QueryInfoKey(key)`

Returns information about a key, as a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

The result is a tuple of 3 items :

In- dex	Signification
0	An integer giving the number of sub keys this key has.
1	An integer giving the number of values this key has.
2	An integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.

`winreg.QueryValue(key, sub_key)`

Retrieves the unnamed value for a key, as a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is *None* or empty, the function retrieves the value set by the *SetValue()* method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a NULL name. But the underlying API call doesn't return the type, so always use `QueryValueEx()` if possible.

`winreg.QueryValueEx(key, value_name)`

Retrieves the type and data for a specified value name associated with an open registry key.

key is an already open key, or one of the predefined `HKEY_* constants`.

value_name is a string indicating the value to query.

The result is a tuple of 2 items :

Index	Signification
0	The value of the registry item.
1	An integer giving the registry type for this value (see table in docs for <code>SetValueEx()</code>)

`winreg.SaveKey(key, file_name)`

Saves the specified key, and all its subkeys to the specified file.

key is an already open key, or one of the predefined `HKEY_* constants`.

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the `LoadKey()` method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the `SeBackupPrivilege` security privilege. Note that privileges are different than permissions – see the [Conflicts Between User Rights and Permissions documentation](#) for more details.

This function passes `NULL` for *security_attributes* to the API.

`winreg.SetValue(key, sub_key, type, value)`

Associates a value with a specified key.

key is an already open key, or one of the predefined `HKEY_* constants`.

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be `REG_SZ`, meaning only strings are supported. Use the `SetValueEx()` function for support for other data types.

value is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the `SetValue` function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with `KEY_SET_VALUE` access.

`winreg.SetValueEx(key, value_name, reserved, type, value)`

Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined `HKEY_* constants`.

value_name is a string that names the subkey with which the value is associated.

reserved can be anything – zero is always passed to the API.

type is an integer that specifies the type of the data. See [Value Types](#) for the available types.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the *key* parameter must have been opened with `KEY_SET_VALUE` access.

To open the key, use the `CreateKey()` or `OpenKey()` methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

`winreg.DisableReflectionKey(key)`

Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined `HKEY_* constants`.

Will generally raise `NotImplemented` if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

`winreg.EnableReflectionKey(key)`

Restores registry reflection for the specified disabled key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise *NotImplemented* if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

`winreg.QueryReflectionKey(key)`

Determines the reflection state for the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Returns `True` if reflection is disabled.

Will generally raise *NotImplemented* if executed on a 32-bit operating system.

34.3.2 Constantes

The following constants are defined for use in many `_winreg` functions.

HKEY_* Constants

`winreg.HKEY_CLASSES_ROOT`

Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

`winreg.HKEY_CURRENT_USER`

Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

`winreg.HKEY_LOCAL_MACHINE`

Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

`winreg.HKEY_USERS`

Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

`winreg.HKEY_PERFORMANCE_DATA`

Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

`winreg.HKEY_CURRENT_CONFIG`

Contains information about the current hardware profile of the local computer system.

`winreg.HKEY_DYN_DATA`

This key is not used in versions of Windows after 98.

Access Rights

For more information, see [Registry Key Security and Access](#).

`winreg.KEY_ALL_ACCESS`

Combines the `STANDARD_RIGHTS_REQUIRED`, `KEY_QUERY_VALUE`, `KEY_SET_VALUE`, `KEY_CREATE_SUB_KEY`, `KEY_ENUMERATE_SUB_KEYS`, `KEY_NOTIFY`, and `KEY_CREATE_LINK` access rights.

`winreg.KEY_WRITE`

Combines the `STANDARD_RIGHTS_WRITE`, `KEY_SET_VALUE`, and `KEY_CREATE_SUB_KEY` access rights.

`winreg.KEY_READ`

Combines the `STANDARD_RIGHTS_READ`, `KEY_QUERY_VALUE`, `KEY_ENUMERATE_SUB_KEYS`, and `KEY_NOTIFY` values.

`winreg.KEY_EXECUTE`

Equivalent to `KEY_READ`.

`winreg.KEY_QUERY_VALUE`

Required to query the values of a registry key.

`winreg.KEY_SET_VALUE`

Required to create, delete, or set a registry value.

`winreg.KEY_CREATE_SUB_KEY`

Required to create a subkey of a registry key.

`winreg.KEY_ENUMERATE_SUB_KEYS`

Required to enumerate the subkeys of a registry key.

`winreg.KEY_NOTIFY`

Required to request change notifications for a registry key or for subkeys of a registry key.

`winreg.KEY_CREATE_LINK`

Reserved for system use.

64-bit Specific

For more information, see [Accessing an Alternate Registry View](#).

`winreg.KEY_WOW64_64KEY`

Indicates that an application on 64-bit Windows should operate on the 64-bit registry view.

`winreg.KEY_WOW64_32KEY`

Indicates that an application on 64-bit Windows should operate on the 32-bit registry view.

Value Types

For more information, see [Registry Value Types](#).

`winreg.REG_BINARY`

Binary data in any form.

`winreg.REG_DWORD`

32-bit number.

`winreg.REG_DWORD_LITTLE_ENDIAN`

A 32-bit number in little-endian format. Equivalent to `REG_DWORD`.

`winreg.REG_DWORD_BIG_ENDIAN`

A 32-bit number in big-endian format.

`winreg.REG_EXPAND_SZ`

Null-terminated string containing references to environment variables (%PATH%).

`winreg.REG_LINK`

A Unicode symbolic link.

`winreg.REG_MULTI_SZ`

A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

`winreg.REG_NONE`

No defined value type.

`winreg.REG_QWORD`

A 64-bit number.

Nouveau dans la version 3.6.

`winreg.REG_QWORD_LITTLE_ENDIAN`

A 64-bit number in little-endian format. Equivalent to `REG_QWORD`.

Nouveau dans la version 3.6.

`winreg.REG_RESOURCE_LIST`

A device-driver resource list.

`winreg.REG_FULL_RESOURCE_DESCRIPTOR`

A hardware setting.

`winreg.REG_RESOURCE_REQUIREMENTS_LIST`

A hardware resource list.

`winreg.REG_SZ`

A null-terminated string.

34.3.3 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the `Close()` method on the object, or the `CloseKey()` function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for `__bool__()` – thus

```
if handle:
    print("Yes")
```

will print `Yes` if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in `int()` function), in which case the underlying Windows handle value is returned. You can also use the `Detach()` method to return the integer handle, and also disconnect the Windows handle from the handle object.

`PyHKEY.Close()`

Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

`PyHKEY.Detach()`

Detaches the Windows handle from the handle object.

The result is an integer that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

`PyHKEY.__enter__()`

`PyHKEY.__exit__(*exc_info)`

The `HKEY` object implements `__enter__()` and `__exit__()` and thus supports the context protocol for the `with` statement :

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ... # work with key
```

will automatically close `key` when control leaves the `with` block.

34.4 winsound — Sound-playing interface for Windows

The `winsound` module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

`winsound.Beep(frequency, duration)`

Beep the PC's speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, `RuntimeError` is raised.

`winsound.PlaySound(sound, flags)`

Call the underlying `PlaySound()` function from the Platform API. The *sound* parameter may be a filename, a system sound alias, audio data as a *bytes-like object*, or `None`. Its interpretation depends on the value of *flags*, which can be a bitwise ORed combination of the constants described below. If the *sound* parameter is `None`, any currently playing waveform sound is stopped. If the system indicates an error, `RuntimeError` is raised.

`winsound.MessageBeep(type=MB_OK)`

Call the underlying `MessageBeep()` function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are `-1`, `MB_ICONASTERISK`, `MB_ICONEXCLAMATION`, `MB_ICONHAND`, `MB_ICONQUESTION`, and `MB_OK`, all described below. The value `-1` produces a « simple beep »; this is the final fallback if a sound cannot be played otherwise. If the system indicates an error, `RuntimeError` is raised.

`winsound.SND_FILENAME`

The *sound* parameter is the name of a WAV file. Do not use with `SND_ALIAS`.

`winsound.SND_ALIAS`

The *sound* parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless `SND_NODEFAULT` is also specified. If no default sound is registered, raise `RuntimeError`. Do not use with `SND_FILENAME`.

All Win32 systems support at least the following; most systems support many more :

<i>PlaySound()</i> <i>name</i>	Corresponding Control Panel Sound name
'SystemAsterisk'	Asterisk
'SystemExclamation'	Exclamation
'SystemExit'	Exit Windows
'SystemHand'	Critical Stop
'SystemQuestion'	Question

Par exemple :

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.SND_LOOP

Play the sound repeatedly. The *SND_ASYNC* flag must also be used to avoid blocking. Cannot be used with *SND_MEMORY*.

winsound.SND_MEMORY

The *sound* parameter to *PlaySound()* is a memory image of a WAV file, as a *bytes-like object*.

Note : This module does not support playing from a memory image asynchronously, so a combination of this flag and *SND_ASYNC* will raise *RuntimeError*.

winsound.SND_PURGE

Stop playing all instances of the specified sound.

Note : This flag is not supported on modern Windows platforms.

winsound.SND_ASYNC

Return immediately, allowing sounds to play asynchronously.

winsound.SND_NODEFAULT

If the specified sound cannot be found, do not play the system default sound.

winsound.SND_NOSTOP

Do not interrupt sounds currently playing.

winsound.SND_NOWAIT

Return immediately if the sound driver is busy.

Note : This flag is not supported on modern Windows platforms.

winsound.MB_ICONASTERISK

Play the SystemDefault sound.

winsound.MB_ICONEXCLAMATION

Play the SystemExclamation sound.

winsound.MB_ICONHAND

Play the SystemHand sound.

`winsound.MB_ICONQUESTION`

Play the `SystemQuestion` sound.

`winsound.MB_OK`

Play the `SystemDefault` sound.

Services spécifiques à Unix

Les modules décrits dans ce chapitre fournissent des interfaces aux fonctionnalités propres au système d'exploitation Unix ou, dans certains cas, à certaines de ses variantes, en voici un aperçu :

35.1 `posix` — Les appels système POSIX les plus courants

Ce module permet d'accéder aux fonctionnalités du système d'exploitation normalisés par le Standard C et le Standard POSIX (une interface Unix habilement déguisée).

Ne pas importer ce module directement. À la place, importer le module `os`, qui fournit une version *portable* de cette interface. Sous Unix, le module `os` fournit un sur-ensemble de l'interface `posix`. Sous les systèmes d'exploitation non Unix le module `posix` n'est pas disponible, mais un sous ensemble est toujours disponible via l'interface `os`. Une fois que `os` est importé, il n'y a aucune perte de performance à l'utiliser à la place de `posix`. De plus, `os` fournit des fonctionnalités supplémentaires, telles que l'appel automatique de `putenv()` lorsqu'une entrée dans `os.environ` est modifiée.

Les erreurs sont signalées comme des exceptions ; les exceptions habituelles sont données pour les erreurs de type, tandis que les erreurs signalées par les appels système lèvent une erreur `OSError`.

35.1.1 Prise en charge de gros fichiers

De nombreux systèmes d'exploitation (y compris AIX, HP-UX, Irix et Solaris) prennent en charge les fichiers d'une taille supérieure à 2 Go malgré que le compilateur C utilise des types `int` et `long` d'une longueur de 32 bit. Ceci est généralement accompli en définissant un nouveau type de 64 bit. Ces fichiers sont parfois appelés fichiers volumineux.

La prise en charge des fichiers volumineux dans Python est activée lorsque la taille de `off_t` est supérieure à `long` et que le type `long long` est disponible et est au moins aussi grand que `off_t`. Il peut-être nécessaire de configurer et compiler Python avec certaines options pour activer ce mode. Par exemple, il est activé par défaut avec les versions récentes d'Irix, mais avec Solaris 2.6 et 2.7 vous avez besoin de faire quelque chose comme :

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \  
./configure
```

Sur les systèmes Linux capable de supporter les fichiers volumineux, cela pourrait fonctionner :

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \  
./configure
```

35.1.2 Contenu du Module

En plus des nombreuses fonctions décrites dans la documentation du module `os`, `posix` possède les éléments suivants :

`posix.envIRON`

Un dictionnaire représentant les variables d'environnement au moment où l'interpréteur a été lancé. Les clés et les valeurs sont des *bytes* sous Unix et des *str* sous Windows. Par exemple, `environ[b'HOME']` (`environ['HOME']` dans Windows) est le chemin de votre dossier d'accueil, équivalent à `getenv("HOME")` en C.

Modifier ce dictionnaire n'affecte pas les variables d'environnements fournis par `execv()`, `popen()` ou `system()` ; Si vous avez besoin de changer l'environnement, passer le paramètre `environ` à `execve()` ou ajouter les assignations de variables et les *export* à la commande à exécuter via `system()` ou `popen()`.

Modifié dans la version 3.2 : Sous Unix, les clés et les valeurs sont des octets.

Note : Le module `os` fournit une implémentation alternative à `environ` qui met à jour l'environnement en cas de modification. Notez également que la mise à jour de `os.environ` rendra ce dictionnaire obsolète. Il est recommandé d'utiliser le module `os` au lieu du module `posix` dans ce cas-ci.

35.2 `pwd` — The password database

This module provides access to the Unix user account and password database. It is available on all Unix versions.

Password database entries are reported as a tuple-like object, whose attributes correspond to the members of the `passwd` structure (Attribute field below, see `<pwd.h>`) :

Index	Attribut	Signification
0	<code>pw_name</code>	Nom d'utilisateur
1	<code>pw_passwd</code>	Optional encrypted password
2	<code>pw_uid</code>	Numerical user ID
3	<code>pw_gid</code>	Numerical group ID
4	<code>pw_gecos</code>	User name or comment field
5	<code>pw_dir</code>	Répertoire d'accueil de l'utilisateur
6	<code>pw_shell</code>	User command interpreter

The uid and gid items are integers, all others are strings. `KeyError` is raised if the entry asked for cannot be found.

Note : In traditional Unix the field `pw_passwd` usually contains a password encrypted with a DES derived algorithm (see module `crypt`). However most modern unices use a so-called *shadow password* system. On those unices the `pw_passwd` field only contains an asterisk (`'*'`) or the letter `'x'` where the encrypted password is stored in a file `/etc/shadow`

which is not world readable. Whether the `pw_passwd` field contains anything useful is system-dependent. If available, the `spwd` module should be used where access to the encrypted password is required.

It defines the following items :

`pwd.getpwuid (uid)`

Return the password database entry for the given numeric user ID.

`pwd.getpwnam (name)`

Return the password database entry for the given user name.

`pwd.getpwall ()`

Return a list of all available password database entries, in arbitrary order.

Voir aussi :

Module `grp` Interface pour la base de données des groupes, similaire à celle-ci.

Module `spwd` An interface to the shadow password database, similar to this.

35.3 spwd — La base de données de mots de passe *shadow*

Ce module permet d'accéder à la base de données UNIX de mots de passe *shadow*. Elle est disponible sur différentes versions d'UNIX.

Vous devez disposer des droits suffisants pour accéder à la base de données de mots de passe *shadow* (cela signifie généralement que vous devez être *root*).

Les entrées de la base de données de mots de passe *shadow* sont renvoyées comme un objet semblable à un tuple, dont les attributs correspondent aux membres de la structure `spwd` (champ attribut ci-dessous, voir `<shadow.h>`) :

In-dex	Attribut	Signification
0	<code>sp_namp</code>	Nom d'utilisateur
1	<code>sp_pwdp</code>	Mot de passe haché
2	<code>sp_lstchg</code>	Date du dernier changement
3	<code>sp_min</code>	Nombre minimal de jours entre les modifications
4	<code>sp_max</code>	Nombre maximal de jours entre les modifications
5	<code>sp_warn</code>	Nombre de jours avant l'expiration du mot de passe pendant lequel l'utilisateur doit être prévenu
6	<code>sp_inact</code>	Nombre de jours avant la désactivation du compte, suite à l'expiration du mot de passe
7	<code>sp_expire</code>	Date à laquelle le compte expire, en nombre de jours depuis le 1er janvier 1970
8	<code>sp_flag</code>	Réservé

Les champs `sp_namp` et `sp_pwdp` sont des chaînes de caractères, tous les autres sont des entiers. `KeyError` est levé si l'entrée demandée est introuvable.

Les fonctions suivantes sont définies :

`spwd.getspnam (name)`

Renvoie l'entrée de base de données de mot de passe *shadow* pour le nom d'utilisateur donné.

Modifié dans la version 3.6 : Lève une `PermissionError` au lieu d'une `KeyError` si l'utilisateur n'a pas les droits suffisants.

`spwd.getspall()`

Renvoie une liste de toutes les entrées de la base de données de mots de passe *shadow*, dans un ordre arbitraire.

Voir aussi :

Module *grp* Interface pour la base de données des groupes, similaire à celle-ci.

Module *pwd* Interface pour la base de données (normale) des mots de passe, semblable à ceci.

35.4 grp — The group database

This module provides access to the Unix group database. It is available on all Unix versions.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the `group` structure (Attribute field below, see `<pwd.h>`) :

Index	Attribut	Signification
0	<code>gr_name</code>	the name of the group
1	<code>gr_passwd</code>	the (encrypted) group password ; often empty
2	<code>gr_gid</code>	the numerical group ID
3	<code>gr_mem</code>	all the group member's user names

The `gid` is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information. Also note that a `gr_name` that starts with a `+` or `-` is likely to be a YP/NIS reference and may not be accessible via `getgrnam()` or `getgrgid().`)

It defines the following items :

`grp.getgrgid(gid)`

Return the group database entry for the given numeric group ID. `KeyError` is raised if the entry asked for cannot be found.

Obsolète depuis la version 3.6 : Since Python 3.6 the support of non-integer arguments like floats or strings in `getgrgid()` is deprecated.

`grp.getgrnam(name)`

Return the group database entry for the given group name. `KeyError` is raised if the entry asked for cannot be found.

`grp.getgrall()`

Return a list of all available group entries, in arbitrary order.

Voir aussi :

Module *pwd* An interface to the user database, similar to this.

Module *spwd* An interface to the shadow password database, similar to this.

35.5 `crypt` — Function to check Unix passwords

Source code : [Lib/crypt.py](#)

This module implements an interface to the `crypt(3)` routine, which is a one-way hash function based upon a modified DES algorithm; see the Unix man page for further details. Possible uses include storing hashed passwords so you can check passwords without storing the actual password, or attempting to crack Unix passwords with a dictionary.

Notice that the behavior of this module depends on the actual implementation of the `crypt(3)` routine in the running system. Therefore, any extensions available on the current implementation will also be available on this module.

35.5.1 Hashing Methods

Nouveau dans la version 3.3.

The `crypt` module defines the list of hashing methods (not all methods are available on all platforms) :

`crypt.METHOD_SHA512`

A Modular Crypt Format method with 16 character salt and 86 character hash. This is the strongest method.

`crypt.METHOD_SHA256`

Another Modular Crypt Format method with 16 character salt and 43 character hash.

`crypt.METHOD_MD5`

Another Modular Crypt Format method with 8 character salt and 22 character hash.

`crypt.METHOD_CRYPT`

The traditional method with a 2 character salt and 13 characters of hash. This is the weakest method.

35.5.2 Module Attributes

Nouveau dans la version 3.3.

`crypt.methods`

A list of available password hashing algorithms, as `crypt.METHOD_*` objects. This list is sorted from strongest to weakest.

35.5.3 Module Functions

The `crypt` module defines the following functions :

`crypt.crypt(word, salt=None)`

`word` will usually be a user's password as typed at a prompt or in a graphical interface. The optional `salt` is either a string as returned from `mk salt()`, one of the `crypt.METHOD_*` values (though not all may be available on all platforms), or a full encrypted password including salt, as returned by this function. If `salt` is not provided, the strongest method will be used (as returned by `methods()`).

Checking a password is usually done by passing the plain-text password as `word` and the full results of a previous `crypt()` call, which should be the same as the results of this call.

`salt` (either a random 2 or 16 character string, possibly prefixed with `$digit$` to indicate the method) which will be used to perturb the encryption algorithm. The characters in `salt` must be in the set `[./a-zA-Z0-9]`, with the exception of Modular Crypt Format which prefixes a `$digit$`.

Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt.

Since a few `crypt(3)` extensions allow different values, with different sizes in the *salt*, it is recommended to use the full cryptéd password as salt when checking for a password.

Modifié dans la version 3.3 : Accept `crypt.METHOD_*` values in addition to strings for *salt*.

`crypt.mksalt` (*method=None*)

Return a randomly generated salt of the specified method. If no *method* is given, the strongest method available as returned by `methods()` is used.

The return value is a string either of 2 characters in length for `crypt.METHOD_CRYPT`, or 19 characters starting with `$digit$` and 16 random characters from the set `[./a-zA-Z0-9]`, suitable for passing as the *salt* argument to `crypt()`.

Nouveau dans la version 3.3.

35.5.4 Exemples

A simple example illustrating typical use (a constant-time comparison operation is needed to limit exposure to timing attacks. `hmac.compare_digest()` is suitable for this purpose) :

```
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
    cryptpasswd = pwd.getpwnam(username)[1]
    if cryptpasswd:
        if cryptpasswd == 'x' or cryptpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptpasswd), cryptpasswd)
    else:
        return True
```

To generate a hash of a password using the strongest available method and check it against the original :

```
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
if not compare_hash(hashed, crypt.crypt(plaintext, hashed)):
    raise ValueError("hashed version doesn't validate against original")
```

35.6 `termios` — Le style POSIX le contrôle TTY

Ce module fournit une interface aux appels POSIX pour le contrôle des E/S TTY. Pour une description complète de ces appels, voir la page du manuel UNIX `termios(3)`. C'est disponible uniquement pour les versions Unix qui supportent le style POSIX `termios` et les contrôles d'entrées/sorties TTY configurés à l'installation.

Toutes les fonctions de ce module prennent un descripteur de fichier *fd* comme premier argument. Ça peut être un descripteur de fichiers entier, tel que le retourne `sys.stdin.fileno()`, ou un *file object*, tel que `sys.stdin`.

Ce module définit aussi toutes les constantes nécessaires pour travailler avec les fonctions fournies ici ; elles ont les mêmes noms que leurs équivalents en C. Pour plus d'informations sur l'utilisation de ces terminaux, veuillez vous référer à votre documentation système.

Le module définit les fonctions suivantes :

`termios.tcgetattr(fd)`

Retourne une liste contenant les attributs TTY pour le descripteur de fichier *fd*, tel que : [*iflag*, *oflag*, *cflag*, *lflag*, *ispeed*, *ospeed*, *cc*] où *cc* est une liste de caractères spéciaux TTY (chacun est une chaîne de caractère de longueur 1, à l'exception des éléments ayant les indices *VMIN* et *VTIME*, ceux-ci sont alors des entiers quand ces champs sont définis). L'interprétation des options (*flags* en anglais) et des vitesses ainsi que l'indexation dans le tableau *cc* doit être fait en utilisant les constantes symboliques définies dans le module *termios*.

`termios.tcsetattr(fd, when, attributes)`

Définit les attributs TTY pour le descripteur de fichiers *fd* à partir des *attributes*, qui est une liste comme celle retournée par *tcgetattr()*. L'argument *when* détermine quand les attributs sont changés : *TCSANOW* pour un changement immédiat, *TCSADRAIN* pour un changement après la transmission de toute sortie en file d'attente, ou *TCSAFLUSH* pour un changement après avoir transmis toute sortie en file d'attente et rejeté toutes entrées en file d'attente.

`termios.tcsendbreak(fd, duration)`

Send a break on file descriptor *fd*. A zero *duration* sends a break for 0.25–0.5 seconds ; a nonzero *duration* has a system dependent meaning.

`termios.tcdrain(fd)`

Attends que toutes les sorties écrites dans le descripteur de fichier *fd* soient transmises.

`termios.tcflush(fd, queue)`

Vide la queue de données du descripteur de fichier *fd*. Le sélecteur *queue* précise la queue : *TCIFLUSH* pour la queue des entrées, *TCOFLUSH* pour la queue des sorties, ou *TCIOFLUSH* pour les deux queues.

`termios.tcflow(fd, action)`

Suspend ou reprends l'entrée ou la sortie du descripteur de fichier *fd*. L'argument *action* peut être *TCOOFF* pour suspendre la sortie, *TCOON* pour relancer la sortie, *TCIOFF* pour suspendre l'entrée, ou *TCION* pour relancer l'entrée.

Voir aussi :

Le module *tty* Fonctions utiles pour les opérations de contrôle communes dans le terminal.

35.6.1 Exemple

Voici une fonction qui demande à l'utilisateur d'entrer un mot de passe sans l'afficher. Remarquez la technique qui consiste à séparer un appel à *tcgetattr()* et une instruction *try... finally* pour s'assurer que les anciens attributs *tty* soient restaurés tels quels quoi qu'il arrive :

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
```

(suite sur la page suivante)

```
termios.tcsetattr(fd, termios.TCSADRAIN, old)
return passwd
```

35.7 `tty` — Fonctions de gestion du terminal

Code source : [Lib/tty.py](#)

Le module `tty` expose des fonctions permettant de mettre le `tty` en mode `cbreak` ou `raw`.

Puisqu'il a besoin du module `termios`, il ne fonctionnera que sur Unix.

Le module `tty` définit les fonctions suivantes :

`tty.setraw(fd, when=termios.TCSAFLUSH)`

Définit le mode du descripteur de fichier `fd` à `raw`. Par défaut, `when` vaut `termios.TCSAFLUSH`, et est passé à `termios.tcsetattr()`.

`tty.setcbreak(fd, when=termios.TCSAFLUSH)`

Définit le mode du descripteur de fichier `fd` à `cbreak`. `when` vaut `termios.TCSAFLUSH` par défaut, et est passé à `termios.tcsetattr()`.

Voir aussi :

Module `termios` Interface bas niveau de gestion du terminal.

35.8 `pty` — Outils de manipulation de pseudo-terminaux

Code source : [Lib/pty.py](#)

Le module `pty` expose des fonctions de manipulation de pseudo-terminaux, il permet d'écrire un programme capable de démarrer un autre processus, d'écrire et de lire depuis son terminal.

La gestion de pseudo-terminaux étant très dépendante de la plateforme, ce code ne gère que Linux. (Code supposé fonctionner sur d'autres plateformes, mais sans avoir été testé).

Le module `pty` expose les fonctions suivantes :

`pty.fork()`

Fork. Connecte le terminal contrôlé par le fils à un pseudo-terminal. La valeur renvoyée est `(pid, fd)`. Notez que le fils obtient 0 comme `pid` et un `fd` non valide. Le parent obtient le `pid` du fils, et `fd` un descripteur de fichier connecté à un terminal contrôlé par le fils (et donc à l'entrée et la sortie standard du fils).

`pty.openpty()`

Ouvre une nouvelle paire de pseudo-terminaux, en utilisant si possible `os.openpty()`, ou du code émulant la fonctionnalité pour des systèmes Unix génériques. Renvoie une paire de descripteurs de fichier (`master`, `slave`), pour le maître et pour l'esclave respectivement.

`pty.spawn(argv[, master_read[, stdin_read]])`

Crée un nouveau processus et connecte son terminal aux entrées/sorties standard du processus courant. C'est typiquement utilisé pour duper les programmes insistant sur le fait de lire depuis leur terminal.

Les fonctions `master_read` et `stdin_read` doivent être des fonctions lisant sur un descripteur de fichier. Par défaut elles lisent 1024 octets à chaque fois qu'elles sont appelées.

Modifié dans la version 3.4 : `spawn()` renvoie maintenant la valeur renvoyée par `os.waitpid()` sur le processus fils.

35.8.1 Exemple

Le programme suivant se comporte comme la commande Unix `script(1)`, utilisant un pseudo-terminal pour enregistrer toutes les entrées et sorties d'une session dans un fichier `typescript`.

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

35.9 fcntl — The fcntl and ioctl system calls

This module performs file control and I/O control on file descriptors. It is an interface to the `fcntl()` and `ioctl()` Unix routines. For a complete description of these calls, see `fcntl(2)` and `ioctl(2)` Unix manual pages.

All functions in this module take a file descriptor `fd` as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or an `io.IOBase` object, such as `sys.stdin` itself, which provides a `fileno()` that returns a genuine file descriptor.

Modifié dans la version 3.3 : Operations in this module used to raise an `IOError` where they now raise an `OSError`.

Le module définit les fonctions suivantes :

`fcntl.fcntl(fd, cmd, arg=0)`

Perform the operation `cmd` on file descriptor `fd` (file objects providing a `fileno()` method are accepted as well). The values used for `cmd` are operating system dependent, and are available as constants in the `fcntl` module, using

the same names as used in the relevant C header files. The argument *arg* can either be an integer value, or a *bytes* object. With an integer value, the return value of this function is the integer return value of the C `fcntl()` call. When the argument is bytes it represents a binary structure, e.g. created by `struct.pack()`. The binary data is copied to a buffer whose address is passed to the C `fcntl()` call. The return value after a successful call is the contents of the buffer, converted to a *bytes* object. The length of the returned object will be the same as the length of the *arg* argument. This is limited to 1024 bytes. If the information returned in the buffer by the operating system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption. If the `fcntl()` fails, an *OSError* is raised.

`fcntl.ioctl(fd, request, arg=0, mutate_flag=True)`

This function is identical to the `fcntl()` function, except that the argument handling is even more complicated. The *request* parameter is limited to values that can fit in 32-bits. Additional constants of interest for use as the *request* argument can be found in the `termios` module, under the same names as used in the relevant C header files.

The parameter *arg* can be one of an integer, an object supporting the read-only buffer interface (like *bytes*) or an object supporting the read-write buffer interface (like *bytearray*).

In all but the last case, behaviour is as for the `fcntl()` function.

If a mutable buffer is passed, then the behaviour is determined by the value of the *mutate_flag* parameter.

If it is false, the buffer's mutability is ignored and behaviour is as for a read-only buffer, except that the 1024 byte limit mentioned above is avoided – so long as the buffer you pass is at least as long as what the operating system wants to put there, things should work.

If *mutate_flag* is true (the default), then the buffer is (in effect) passed to the underlying `ioctl()` system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the `ioctl()`. This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to `ioctl()` and copied back into the supplied buffer.

If the `ioctl()` fails, an *OSError* exception is raised.

Un exemple :

```
>>> import array, fcntl, struct, termios, os
>>> os.getpggrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPRG, " "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPRG, buf, 1)
0
>>> buf
array('h', [13341])
```

`fcntl.flock(fd, operation)`

Perform the lock operation *operation* on file descriptor *fd* (file objects providing a `fileno()` method are accepted as well). See the Unix manual `flock(2)` for details. (On some systems, this function is emulated using `fcntl()`.)

If the `flock()` fails, an *OSError* exception is raised.

`fcntl.lockf(fd, cmd, len=0, start=0, whence=0)`

This is essentially a wrapper around the `fcntl()` locking calls. *fd* is the file descriptor of the file to lock or unlock, and *cmd* is one of the following values :

- LOCK_UN – unlock
- LOCK_SH – acquire a shared lock
- LOCK_EX – acquire an exclusive lock

When *cmd* is LOCK_SH or LOCK_EX, it can also be bitwise ORed with LOCK_NB to avoid blocking on lock acquisition. If LOCK_NB is used and the lock cannot be acquired, an *OSError* will be raised and the exception will have an *errno* attribute set to EACCES or EAGAIN (depending on the operating system ; for portability, check

for both values). On at least some systems, `LOCK_EX` can only be used if the file descriptor refers to a file opened for writing.

len is the number of bytes to lock, *start* is the byte offset at which the lock starts, relative to *whence*, and *whence* is as with `io.IOBase.seek()`, specifically :

- 0 – relative to the start of the file (`os.SEEK_SET`)
- 1 – relative to the current buffer position (`os.SEEK_CUR`)
- 2 – relative to the end of the file (`os.SEEK_END`)

The default for *start* is 0, which means to start at the beginning of the file. The default for *len* is 0 which means to lock to the end of the file. The default for *whence* is also 0.

Examples (all on a SVR4 compliant system) :

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable *rv* will hold an integer value ; in the second example it will hold a *bytes* object. The structure lay-out for the *lockdata* variable is system dependent — therefore using the `flock()` call may be better.

Voir aussi :

Module `os` If the locking flags `O_SHLOCK` and `O_EXLOCK` are present in the `os` module (on BSD only), the `os.open()` function provides an alternative to the `lockf()` and `flock()` functions.

35.10 pipes — Interface au *pipelines* shell

Code source : [Lib/pipes.py](#)

Le module `pipes` définit une classe permettant d'abstraire le concept de *pipeline* — une séquence de convertisseurs d'un fichier vers un autre.

Du fait que le module utilise les lignes de commandes `/bin/sh`, un shell POSIX ou compatible est requis pour `os.system()` et `os.popen()`.

Le module `pipes` définit la classe suivante :

class `pipes.Template`
Une abstraction d'un *pipeline*.

Exemple :

```
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

35.10.1 L'Objet *Template*

Les méthodes de l'objet *Template* :

`Template.reset()`

Réinitialise un modèle de *pipeline* à son état initial.

`Template.clone()`

Renvoie un nouveau modèle de *pipeline*, équivalent.

`Template.debug(flag)`

Si *flag* est vrai, active le débogage. Sinon, le désactive. Quand le débogage est actif, les commandes à exécuter seront affichées et le shell est pourvu de la commande `set -x` afin d'être plus verbeux.

`Template.append(cmd, kind)`

Ajoute une nouvelle action à la fin. La variable *cmd* doit être une commande *bourne shell* valide. La variable *kind* est composée de deux lettres.

La première lettre peut soit être '-' (qui signifie que la commande lit sa sortie standard), soit 'f' (qui signifie que la commande lit un fichier donné par la ligne de commande), soit '.' (qui signifie que la commande ne lit pas d'entrée, et donc doit être la première.)

De même, la seconde lettre peut soit être '-' (qui signifie que la commande écrit sur la sortie standard), soit 'f' (qui signifie que la commande écrit sur un fichier donné par la ligne de commande), soit '.' (qui signifie que la commande n'écrit rien, et donc doit être la dernière.)

`Template.prepend(cmd, kind)`

Ajoute une nouvelle action au début. Voir `append()` pour plus d'explications sur les arguments.

`Template.open(file, mode)`

Renvoie un objet *file-like*, ouvert à *file*, mais permettant d'écrire vers le *pipeline* ou de lire depuis celui-ci. À noter que seulement un des deux ("r" ou "w") peut être donné.

`Template.copy(infile, outfile)`

Copie *infile* vers *outfile* au travers du *pipe*.

35.11 resource — Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

An `OSError` is raised on syscall failure.

exception `resource.error`

A deprecated alias of `OSError`.

Modifié dans la version 3.3 : Following **PEP 3151**, this class was made an alias of `OSError`.

35.11.1 Resource Limits

Resources usage can be limited using the `setrlimit()` function described below. Each resource is controlled by a pair of limits : a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the `getrlimit(2)` man page. The resources listed below are supported when the underlying operating system supports them ; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

`resource.RLIM_INFINITY`

Constant used to represent the limit for an unlimited resource.

`resource.getrlimit(resource)`

Returns a tuple (soft, hard) with the current soft and hard limits of *resource*. Raises `ValueError` if an invalid resource is specified, or `error` if the underlying system call fails unexpectedly.

`resource.setrlimit(resource, limits)`

Sets new limits of consumption of *resource*. The *limits* argument must be a tuple (soft, hard) of two integers describing the new limits. A value of `RLIM_INFINITY` can be used to request a limit that is unlimited.

Raises `ValueError` if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of `RLIM_INFINITY` when the hard or system limit for that resource is not unlimited will result in a `ValueError`. A process with the effective UID of super-user can request any valid limit value, including unlimited, but `ValueError` will still be raised if the requested limit exceeds the system imposed limit.

`setrlimit` may also raise `error` if the underlying system call fails.

`resource.prlimit(pid, resource[, limits])`

Combines `setrlimit()` and `getrlimit()` in one function and supports to get and set the resources limits of an arbitrary process. If *pid* is 0, then the call applies to the current process. *resource* and *limits* have the same meaning as in `setrlimit()`, except that *limits* is optional.

When *limits* is not given the function returns the *resource* limit of the process *pid*. When *limits* is given the *resource* limit of the process is set and the former resource limit is returned.

Raises `ProcessLookupError` when *pid* can't be found and `PermissionError` when the user doesn't have `CAP_SYS_RESOURCE` for the process.

Availability : Linux 2.6.36 or later with glibc 2.13 or later

Nouveau dans la version 3.4.

These symbols define resources whose consumption can be controlled using the `setrlimit()` and `getrlimit()` functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for `getrlimit(2)` lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences — symbols not defined for a platform will not be available from this module on that platform.

`resource.RLIMIT_CORE`

The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

`resource.RLIMIT_CPU`

The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a `SIGXCPU` signal is sent to the process. (See the `signal` module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

`resource.RLIMIT_FSIZE`

The maximum size of a file which the process may create.

`resource.RLIMIT_DATA`

The maximum size (in bytes) of the process's heap.

`resource.RLIMIT_STACK`

The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

`resource.RLIMIT_RSS`

The maximum resident set size that should be made available to the process.

`resource.RLIMIT_NPROC`

The maximum number of processes the current process may create.

`resource.RLIMIT_NOFILE`

The maximum number of open file descriptors for the current process.

`resource.RLIMIT_OFILE`

The BSD name for `RLIMIT_NOFILE`.

`resource.RLIMIT_MEMLOCK`

The maximum address space which may be locked in memory.

`resource.RLIMIT_VMEM`

The largest area of mapped memory which the process may occupy.

`resource.RLIMIT_AS`

The maximum area (in bytes) of address space which may be taken by the process.

`resource.RLIMIT_MSGQUEUE`

The number of bytes that can be allocated for POSIX message queues.

Availability : Linux 2.6.8 or later.

Nouveau dans la version 3.4.

`resource.RLIMIT_NICE`

The ceiling for the process's nice level (calculated as $20 - \text{rlim_cur}$).

Availability : Linux 2.6.12 or later.

Nouveau dans la version 3.4.

`resource.RLIMIT_RTPRIO`

The ceiling of the real-time priority.

Availability : Linux 2.6.12 or later.

Nouveau dans la version 3.4.

`resource.RLIMIT_RTTIME`

The time limit (in microseconds) on CPU time that a process can spend under real-time scheduling without making a blocking syscall.

Availability : Linux 2.6.25 or later.

Nouveau dans la version 3.4.

`resource.RLIMIT_SIGPENDING`

The number of signals which the process may queue.

Availability : Linux 2.6.8 or later.

Nouveau dans la version 3.4.

`resource.RLIMIT_SBSIZE`

The maximum size (in bytes) of socket buffer usage for this user. This limits the amount of network memory, and hence the amount of mbufs, that this user may hold at any time.

Availability : FreeBSD 9 or later.

Nouveau dans la version 3.4.

resource.RLIMIT_SWAP

The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id's processes. This limit is enforced only if bit 1 of the `vm.overcommit` sysctl is set. Please see *tuning(7)* for a complete description of this sysctl.

Availability : FreeBSD 9 or later.

Nouveau dans la version 3.4.

resource.RLIMIT_NPTS

The maximum number of pseudo-terminals created by this user id.

Availability : FreeBSD 9 or later.

Nouveau dans la version 3.4.

35.11.2 Resource Usage

These functions are used to retrieve resource usage information :

resource.getrusage(who)

This function returns an object that describes the resources consumed by either the current process or its children, as specified by the *who* parameter. The *who* parameter should be specified using one of the `RUSAGE_*` constants described below.

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running in user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick interval, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields `ru_utime` and `ru_stime` of the return value are floating point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the *getrusage(2)* man page for detailed information about these values. A brief summary is presented here :

Index	Field	Resource
0	<code>ru_utime</code>	time in user mode (float)
1	<code>ru_stime</code>	time in system mode (float)
2	<code>ru_maxrss</code>	maximum resident set size
3	<code>ru_ixrss</code>	shared memory size
4	<code>ru_idrss</code>	unshared memory size
5	<code>ru_isrss</code>	unshared stack size
6	<code>ru_minflt</code>	page faults not requiring I/O
7	<code>ru_majflt</code>	page faults requiring I/O
8	<code>ru_nswap</code>	number of swap outs
9	<code>ru_inblock</code>	block input operations
10	<code>ru_oublock</code>	block output operations
11	<code>ru_msgsnd</code>	messages sent
12	<code>ru_msgrcv</code>	messages received
13	<code>ru_nsignals</code>	signals received
14	<code>ru_nvcsw</code>	voluntary context switches
15	<code>ru_nivcsw</code>	involuntary context switches

This function will raise a *ValueError* if an invalid *who* parameter is specified. It may also raise *error* exception in unusual circumstances.

resource.getpagesize()

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following `RUSAGE_*` symbols are passed to the `getrusage()` function to specify which processes information should be provided for.

`resource.RUSAGE_SELF`

Pass to `getrusage()` to request resources consumed by the calling process, which is the sum of resources used by all threads in the process.

`resource.RUSAGE_CHILDREN`

Pass to `getrusage()` to request resources consumed by child processes of the calling process which have been terminated and waited for.

`resource.RUSAGE_BOTH`

Pass to `getrusage()` to request resources consumed by both the current process and child processes. May not be available on all systems.

`resource.RUSAGE_THREAD`

Pass to `getrusage()` to request resources consumed by the current thread. May not be available on all systems. Nouveau dans la version 3.2.

35.12 nis — Interface à Sun's NIS (pages jaunes)

Le module `nis` est une simple abstraction de la librairie NIS, utile pour l'administration centralisée de plusieurs hôtes.

Du fait que NIS existe seulement sur les systèmes Unix, ce module est seulement disponible pour Unix.

Le module `nis` définit les instructions suivantes :

`nis.match(key, mapname, domain=default_domain)`

Renvoie la valeur correspondante à `key` dans carte `mapname`, ou lève une erreur (`nis.error`) s'il n'y en a pas. Toutes les deux doivent être des chaînes, `key` doit être une chaîne ASCII. La valeur renvoyée est un dictionnaire arbitraire d'octets (pourrait contenir `NULL` et autres joyeusetés).

Notez que `mapname` est vérifié la première fois si c'est un alias d'un autre nom.

L'argument `domain` permet de passer outre le domaine NIS utilisé pour les recherches. Lorsqu'il n'est pas spécifié, recherche est dans le domaine NIS défaut.

`nis.cat(mapname, domain=default_domain)`

Renvoie un dictionnaire qui associe `key` à `value` tel que `match(key, mapname) == value`. Notez que les clés comme les valeurs peuvent contenir des séquences arbitraires d'octets.

Notez que `mapname` est vérifié la première fois si c'est un alias d'un autre nom.

L'argument `domain` permet de passer outre le domaine NIS utilisé pour les recherches. Lorsqu'il n'est pas spécifié, recherche est dans le domaine NIS défaut.

`nis.maps(domain=default_domain)`

Renvoie la liste de toutes les correspondances valides.

L'argument `domain` permet de passer outre le domaine NIS utilisé pour les recherches. Lorsqu'il n'est pas spécifié, recherche est dans le domaine NIS défaut.

`nis.get_default_domain()`

Renvoie le domaine NIS par défaut du système.

Le module `nis` définit les exceptions suivantes :

exception `nis.error`

Une erreur apparaît quand une fonction NIS renvoie un code d'erreur.

35.13 syslog — Unix syslog library routines

This module provides an interface to the Unix `syslog` library routines. Refer to the Unix manual pages for a detailed description of the `syslog` facility.

This module wraps the system `syslog` family of routines. A pure Python library that can speak to a syslog server is available in the `logging.handlers` module as `SysLogHandler`.

Le module définit les fonctions suivantes :

`syslog.syslog(message)`

`syslog.syslog(priority, message)`

Send the string *message* to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a *facility* and a *level*. The optional *priority* argument, which defaults to `LOG_INFO`, determines the message priority. If the facility is not encoded in *priority* using logical-or (`LOG_INFO | LOG_USER`), the value given in the `openlog()` call is used.

If `openlog()` has not been called prior to the call to `syslog()`, `openlog()` will be called with no arguments.

`syslog.openlog([ident[, logoption[, facility]]])`

Logging options of subsequent `syslog()` calls can be set by calling `openlog()`. `syslog()` will call `openlog()` with no arguments if the log is not currently open.

The optional *ident* keyword argument is a string which is prepended to every message, and defaults to `sys.argv[0]` with leading path components stripped. The optional *logoption* keyword argument (default is 0) is a bit field – see below for possible values to combine. The optional *facility* keyword argument (default is `LOG_USER`) sets the default facility for messages which do not have a facility explicitly encoded.

Modifié dans la version 3.2 : In previous versions, keyword arguments were not allowed, and *ident* was required. The default for *ident* was dependent on the system libraries, and often was `python` instead of the name of the python program file.

`syslog.closelog()`

Reset the syslog module values and call the system library `closelog()`.

This causes the module to behave as it does when initially imported. For example, `openlog()` will be called on the first `syslog()` call (if `openlog()` hasn't already been called), and *ident* and other `openlog()` parameters are reset to defaults.

`syslog.setlogmask(maskpri)`

Set the priority mask to *maskpri* and return the previous mask value. Calls to `syslog()` with a priority level not set in *maskpri* are ignored. The default is to log all priorities. The function `LOG_MASK(pri)` calculates the mask for the individual priority *pri*. The function `LOG_UPTO(pri)` calculates the mask for all priorities up to and including *pri*.

The module defines the following constants :

Priority levels (high to low) : `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, `LOG_DEBUG`.

Facilities : `LOG_KERN`, `LOG_USER`, `LOG_MAIL`, `LOG_DAEMON`, `LOG_AUTH`, `LOG_LPR`, `LOG_NEWS`, `LOG_UUCP`, `LOG_CRON`, `LOG_SYSLOG`, `LOG_LOCAL0` to `LOG_LOCAL7`, and, if defined in `<syslog.h>`, `LOG_AUTHPRIV`.

Log options : `LOG_PID`, `LOG_CONS`, `LOG_NDELAY`, and, if defined in `<syslog.h>`, `LOG_ODELAY`, `LOG_NOWAIT`, and `LOG_PERROR`.

35.13.1 Examples

Simple example

A simple set of examples :

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

An example of setting some log options, these would include the process ID in logged messages, and write the messages to the destination facility used for mail logging :

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

Modules remplacés

Les modules documentés ici sont obsolètes et ne sont gardés que pour la rétro-compatibilité. Ils ont été remplacés par d'autres modules.

36.1 `optparse` — Parser for command line options

Code source : [Lib/optparse.py](#)

Obsolète depuis la version 3.2 : The `optparse` module is deprecated and will not be developed further ; development will continue with the `argparse` module.

`optparse` is a more convenient, flexible, and powerful library for parsing command-line options than the old `getopt` module. `optparse` uses a more declarative style of command-line parsing : you create an instance of `OptionParser`, populate it with options, and parse the command line. `optparse` allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using `optparse` in a simple script :

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the « usual thing » on the command-line, for example :

```
<yourscript> --file=outfile -q
```

As it parses the command line, `optparse` sets attributes of the `options` object returned by `parse_args()` based on user-supplied command-line values. When `parse_args()` returns from parsing this command line, `options.filename` will be "outfile" and `options.verbose` will be `False`. `optparse` supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example :

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of

```
<yourscript> -h
<yourscript> --help
```

and `optparse` will print out a brief summary of your script's options :

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet            don't print status messages to stdout
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

36.1.1 Background

`optparse` was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

Terminology

argument a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term « word ».

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read « argument » as « an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]` ».

option an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen (« - ») followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by `optparse`.

Some other option syntaxes that the world has seen include :

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)
- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`

— a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by `optparse`, and they never will be. This is deliberate : the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting VMS, MS-DOS, and/or Windows.

option argument an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With `optparse`, option arguments may either be in a separate argument from their option :

```
-f foo
--file foo
```

or included in the same argument :

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an « optional option arguments » feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous : if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, `optparse` does not support this feature.

argument positionnel something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

required option an option that must be supplied on the command-line ; note that the phrase « required option » is self-contradictory in English. `optparse` doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line :

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for ?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find`, `tar`, and `dd`—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have « required options ». Think about it. If it's required, then it's *not optional* ! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all :

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied : you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

What are positional arguments for ?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter *how* you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI : if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the « Preferences » dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course ; too many options can overwhelm users and make your code much harder to maintain.

36.1.2 Tutoriel

While `optparse` is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any `optparse`-based program.

First, you need to import the `OptionParser` class ; then, early in the main program, create an `OptionParser` instance :

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is :

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell `optparse` what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g. :

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to `OptionParser.add_option()` are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line ; in reality, `optparse` encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct `optparse` to parse your program's command line :

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to `parse_args()`, but that's rarely necessary : by default it uses `sys.argv[1:]`.)

`parse_args()` returns two values :

- `options`, an object containing values for all of your options—e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes : *action*, *type*, *dest* (destination), and *help*. Of these, *action* is the most fundamental.

Understanding option actions

Actions tell *optparse* what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into *optparse*; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell *optparse* to store a value in some variable—for example, take a string from the command line and store it in an attribute of options.

If you don't specify an option action, *optparse* defaults to *store*.

The store action

The most common option action is *store*, which tells *optparse* to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

Par exemple :

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask *optparse* to parse it :

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When *optparse* sees the option string *-f*, it consumes the next argument, *foo.txt*, and stores it in *options.filename*. So, after this call to *parse_args()*, *options.filename* is *"foo.txt"*.

Some other option types supported by *optparse* are *int* and *float*. Here's an option that expects an integer argument :

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is *store*.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option : since *-n42* (one argument) is equivalent to *-n 42* (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

affichera 42.

If you don't specify a type, *optparse* assumes *string*. Combined with the fact that the default action is *store*, that means our first example can be a lot shorter :

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, *optparse* figures out a sensible default from the option strings : if the first long option string is *--foo-bar*, then the default destination is *foo_bar*. If there are no long option strings, *optparse* looks at the first short option string : the default destination for *-f* is *f*.

optparse also includes the built-in *complex* type. Adding types is covered in section *Extending optparse*.

Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen—are quite common. `optparse` supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values—see below.)

When `optparse` encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

Other actions

Some other actions supported by `optparse` are :

- "**store_const**" store a constant value
- "**append**" append this option's argument to a list
- "**count**" increment a counter by one
- "**callback**" call a specified function

These are covered in section [Reference Guide](#), Reference Guide and section [Option Callbacks](#).

Valeurs par défaut

All of the above examples involve setting some variable (the « destination ») when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. `optparse` lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want `optparse` to set `verbose` to `True` unless `-q` is seen, then we can do this :

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent :

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Consider this :

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True` : the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of `OptionParser`, which you can call at any time before calling `parse_args()` :

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

optparse's ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a *help* value for each option, and optionally a short usage message for your whole program. Here's an *OptionParser* populated with user-friendly (documented) options :

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                        "or expert [default: %default]")
```

If *optparse* encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output :

```
Usage: <yourscrip> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose          make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, *optparse* exits after printing the help text.)

There's a lot going on here to help *optparse* generate the best possible help message :

- the script defines its own usage message :

```
usage = "usage: %prog [options] arg1 arg2"
```

optparse expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

If you don't supply a usage string, *optparse* uses a bland but sensible default : `"Usage: %prog [options]"`, which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping—*optparse* takes care of wrapping lines and making the help output look good.
- options that take a value indicate this fact in their automatically-generated help message, e.g. for the « mode » option :

```
-m MODE, --mode=MODE
```

Here, « MODE » is called the meta-variable : it stands for the argument that the user is expected to supply to `-m/--mode`. By default, `optparse` converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want—for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically-generated option description :

```
-f FILE, --filename=FILE
```

This is important for more than just saving space, though : the manually written help text uses the meta-variable `FILE` to clue the user in that there's a connection between the semi-formal syntax `-f FILE` and the informal semantic description « write output to `FILE` ». This is a simple but effective way to make your help text a lot clearer and more useful for end users.

- options that have a default value can include `%default` in the help string—`optparse` will replace it with `str()` of the option's default value. If an option has no default value (or the default value is `None`), `%default` expands to `none`.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An `OptionParser` can contain several option groups, each of which can contain several options.

An option group is obtained using the class `OptionGroup` :

```
class optparse.OptionGroup (parser, title, description=None)
    where
    — parser is the OptionParser instance the group will be inserted in to
    — title is the group title
    — description, optional, is a long description of the group
```

`OptionGroup` inherits from `OptionContainer` (like `OptionParser`) and so the `add_option()` method can be used to add an option to the group.

Once all the options are declared, using the `OptionParser` method `add_option_group()` the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an `OptionGroup` to a parser is easy :

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output :

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose          make lots of noise [default]
  -q, --quiet            be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(suite sur la page suivante)

(suite de la page précédente)

```
Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                Group option.
```

A bit more complete example might involve using more than one group : still extending the previous example :

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

that results in the following output :

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose          make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                Group option.

Debug Options:
  -d, --debug          Print debug information
  -s, --sql            Print all SQL statements executed
  -e                  Print every action done
```

Another interesting method, in particular when working programmatically with option groups is :

`OptionParser.get_option_group(opt_str)`

Return the *OptionGroup* to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there's no such *OptionGroup*, return `None`.

Printing a version string

Similar to the brief usage string, `optparse` can also print a version string for your program. You have to supply the string as the `version` argument to `OptionParser` :

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` is expanded just like it is in usage. Apart from that, `version` can contain anything you like. When you supply it, `optparse` automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo` :

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string :

`OptionParser.print_version(file=None)`

Print the version message for the current program (`self.version`) to *file* (default stdout). As with `print_usage()`, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

`OptionParser.get_version()`

Same as `print_version()` but returns the version string instead of printing it.

How `optparse` handles errors

There are two broad classes of errors that `optparse` has to worry about : programmer errors and user errors. Programmer errors are usually erroneous calls to `OptionParser.add_option()`, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way : raise an exception (either `optparse.OptionError` or `TypeError`) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. `optparse` can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition :

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, `optparse` handles the error the same way : it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer :

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all :

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error()` from your application code.

If *optparse*'s default error-handling behaviour does not suit your needs, you'll need to subclass `OptionParser` and override its `exit()` and/or `error()` methods.

Putting it all together

Here's what *optparse*-based scripts usually look like :

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

36.1.3 Reference Guide

Creating the parser

The first step in using *optparse* is to create an `OptionParser` instance.

class `optparse.OptionParser(...)`

The `OptionParser` constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

usage (default: `"%prog [options]"`) The usage summary to print when your program is run incorrectly or with a help option. When *optparse* prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

option_list (default: `[]`) A list of `Option` objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by `OptionParser` subclasses), but before any version or help options. Deprecated; use `add_option()` after creating the parser instead.

option_class (default : `optparse.Option`) Class to use when adding options to the parser in `add_option()`.

version (default : `None`) A version string to print when the user supplies a version option. If you supply a true value for `version`, `optparse` automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

conflict_handler (default : `"error"`) Specifies what to do when options with conflicting option strings are added to the parser; see section *Conflicts between options*.

description (default : `None`) A paragraph of text giving a brief overview of your program. `optparse` reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

formatter (default : a new `IndentedHelpFormatter`) An instance of `optparse.HelpFormatter` that will be used for printing help text. `optparse` provides two concrete classes for this purpose : `IndentedHelpFormatter` and `TitledHelpFormatter`.

add_help_option (default : `True`) If true, `optparse` will add a help option (with option strings `-h` and `--help`) to the parser.

prog The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

epilog (default : `None`) A paragraph of help text to print after the option help.

Populating the parser

There are several ways to populate the parser with options. The preferred way is by using `OptionParser.add_option()`, as shown in section *Tutorial*. `add_option()` can be called in one of two ways :

- pass it an `Option` instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the `Option` constructor), and it will create the `Option` instance for you

The other alternative is to pass a list of pre-constructed `Option` instances to the `OptionParser` constructor, as in :

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating `Option` instances; currently it is an alias for the `Option` constructor. A future version of `optparse` may split `Option` into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate `Option` directly.)

Defining options

Each `Option` instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an `Option` instance is with the `add_option()` method of `OptionParser`.

```
OptionParser.add_option(option)
OptionParser.add_option(*opt_str, attr=value, ...)
```

To define an option with only a short option string :


```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string :

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is *action*, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, *optparse* raises an `OptionError` exception explaining your mistake.

An option's *action* determines what *optparse* does when it encounters this option on the command-line. The standard option actions hard-coded into *optparse* are :

"store" store this option's argument (default)

"store_const" store a constant value

"store_true" store a true value

"store_false" store a false value

"append" append this option's argument to a list

"append_const" append a constant value to a list

"count" increment a counter by one

"callback" call a specified function

"help" print a usage message including all options and the documentation for them

(If you don't supply an action, the default is "store". For this action, you may also supply *type* and *dest* option attributes; see *Standard option actions*.)

As you can see, most actions involve storing or updating a value somewhere. *optparse* always creates a special object for this, conventionally called `options` (it happens to be an instance of `optparse.Values`). Option arguments (and various other values) are stored as attributes of this object, according to the *dest* (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things *optparse* does is create the `options` object :

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following :

```
-ffoo
-f foo
--file=foo
--file foo
```

then *optparse*, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The *type* and *dest* option attributes are almost as important as *action*, but *action* is the only one that makes sense for *all* options.

Option attributes

The following option attributes may be passed as keyword arguments to `OptionParser.add_option()`. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, `optparse` raises `OptionError`.

`Option.action`

(default : "store")

Determines `optparse`'s behaviour when this option is seen on the command line ; the available options are documented [here](#).

`Option.type`

(default : "string")

The argument type expected by this option (e.g., "string" or "int"); the available option types are documented [here](#).

`Option.dest`

(default : derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells `optparse` where to write it : `dest` names an attribute of the `options` object that `optparse` builds as it parses the command line.

`Option.default`

The value to use for this option's destination if the option is not seen on the command line. See also `OptionParser.set_defaults()`.

`Option.nargs`

(default : 1)

How many arguments of type `type` should be consumed when this option is seen. If > 1, `optparse` will store a tuple of values to `dest`.

`Option.const`

For actions that store a constant value, the constant value to store.

`Option.choices`

For options of type "choice", the list of strings the user may choose from.

`Option.callback`

For options with action "callback", the callable to call when this option is seen. See section [Option Callbacks](#) for detail on the arguments passed to the callable.

`Option.callback_args`

`Option.callback_kwargs`

Additional positional and keyword arguments to pass to `callback` after the four standard callback arguments.

`Option.help`

Help text to print for this option when listing all available options after the user supplies a `help` option (such as `--help`). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `optparse.SUPPRESS_HELP`.

`Option.metavar`

(default : derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section [Tutoriel](#) for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*'s behaviour; a few have required attributes, which you must specify for any option using that action.

- "store" [relevant : *type*, *dest*, *nargs*, *choices*]

The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line; all will be converted according to *type* and stored to *dest* as a tuple. See the *Standard option types* section.

If *choices* is supplied (a list or tuple of strings), the type defaults to "choice".

If *type* is not supplied, it defaults to "string".

If *dest* is not supplied, *optparse* derives a destination from the first long option string (e.g., --foo-bar implies foo_bar). If there are no long option strings, *optparse* derives a destination from the first short option string (e.g., -f implies f).

Example :

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

optparse will set

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- "store_const" [required : *const*; relevant : *dest*]

The value *const* is stored in *dest*.

Example :

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

If --noisy is seen, *optparse* will set

```
options.verbose = 2
```

- "store_true" [relevant : *dest*]

A special case of "store_const" that stores a true value to *dest*.

- "store_false" [relevant : *dest*]

Like "store_true", but stores a false value.

Example :

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [relevant : *type*, *dest*, *nargs*, *choices*]

The option must be followed by an argument, which is appended to the list in *dest*. If no default value for *dest* is supplied, an empty list is automatically created when *optparse* first encounters this option on the command-line. If *nargs* > 1, multiple arguments are consumed, and a tuple of length *nargs* is appended to *dest*.

The defaults for *type* and *dest* are the same as for the "store" action.

Example :

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

If `-t3` is seen on the command-line, `optparse` does the equivalent of :

```
options.tracks = []
options.tracks.append(int("3"))
```

If, a little later on, `--tracks=4` is seen, it does :

```
options.tracks.append(int("4"))
```

The `append` action calls the `append` method on the current value of the option. This means that any default value specified must have an `append` method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values :

```
>>> parser.add_option("--files", action="append", default=['~/mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/mypkg/defaults', 'overrides.mypkg']
```

- `"append_const"` [required : `const`; relevant : `dest`]

Like `"store_const"`, but the value `const` is appended to `dest`; as with `"append"`, `dest` defaults to `None`, and an empty list is automatically created the first time the option is encountered.

- `"count"` [relevant : `dest`]

Increment the integer stored at `dest`. If no default value is supplied, `dest` is set to zero before being incremented the first time.

Exemple :

```
parser.add_option("-v", action="count", dest="verbosity")
```

The first time `-v` is seen on the command line, `optparse` does the equivalent of :

```
options.verbosity = 0
options.verbosity += 1
```

Every subsequent occurrence of `-v` results in

```
options.verbosity += 1
```

- `"callback"` [required : `callback`; relevant : `type`, `nargs`, `callback_args`, `callback_kwargs`]
Call the function specified by `callback`, which is called as

```
func(option, opt_str, value, parser, *args, **kwargs)
```

See section [Option Callbacks](#) for more detail.

- `"help"`

Prints a complete help message for all the options in the current option parser. The help message is constructed from the usage string passed to `OptionParser`'s constructor and the `help` string passed to every option.

If no `help` string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value `optparse.SUPPRESS_HELP`.

`optparse` automatically adds a `help` option to all `OptionParsers`, so you do not normally need to create one.

Exemple :

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)
```

(suite sur la page suivante)

(suite de la page précédente)

```

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)

```

If *optparse* sees either `-h` or `--help` on the command line, it will print something like the following help message to stdout (assuming `sys.argv[0]` is `"foo.py"`):

```

Usage: foo.py [options]

Options:
  -h, --help            Show this help message and exit
  -v                    Be moderately verbose
  --file=FILENAME       Input file to read data from

```

After printing the help message, *optparse* terminates your process with `sys.exit(0)`.

— "version"

Prints the version number supplied to the `OptionParser` to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of `OptionParser`. Generally only relevant if the `version` argument is supplied to the `OptionParser` constructor. As with *help* options, you will rarely create *version* options, since *optparse* automatically adds them when needed.

Standard option types

optparse has five built-in option types: `"string"`, `"int"`, `"choice"`, `"float"` and `"complex"`. If you need to add new option types, see section *Extending optparse*.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type `"int"`) are parsed as follows:

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling `int()` with the appropriate base (2, 8, 10, or 16). If this fails, so will *optparse*, although with a more useful error message.

`"float"` and `"complex"` option arguments are converted directly with `float()` and `complex()`, with similar error-handling.

`"choice"` options are a subtype of `"string"` options. The *choices* option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice()` compares user-supplied option arguments against this master list and raises `OptionValueError` if an invalid string is given.

Analyse des arguments

The whole point of creating and populating an `OptionParser` is to call its `parse_args()` method :

```
(options, args) = parser.parse_args(args=None, values=None)
```

where the input parameters are

args the list of arguments to process (default : `sys.argv[1:]`)

values an `optparse.Values` object to store option arguments in (default : a new instance of `Values`) – if you give an existing object, the option defaults will not be initialized on it

and the return values are

options the same object that was passed in as `values`, or the `optparse.Values` instance created by *optparse*

args the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated `setattr()` calls (roughly one for every option argument stored to an option destination) and returned by `parse_args()`.

If `parse_args()` encounters any errors in the argument list, it calls the `OptionParser`'s `error()` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. `OptionParser` provides several methods to help you out :

`OptionParser.disable_interspersed_args()`

Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, *optparse* normally accepts this syntax :

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call `disable_interspersed_args()`. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

`OptionParser.enable_interspersed_args()`

Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

`OptionParser.get_option(opt_str)`

Returns the `Option` instance with the option string `opt_str`, or `None` if no options have that option string.

`OptionParser.has_option(opt_str)`

Return true if the `OptionParser` has an option with option string `opt_str` (e.g., `-q` or `--verbose`).

`OptionParser.remove_option(opt_str)`

If the *OptionParser* has an option corresponding to `opt_str`, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If `opt_str` does not occur in any option belonging to this *OptionParser*, raises `ValueError`.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings :

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own `OptionParser` subclass with some standard options.)

Every time you add an option, `optparse` checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor :

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call :

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are :

"error" (default) assume option conflicts are a programming error and raise `OptionConflictError`

"resolve" resolve option conflicts intelligently (see below)

As an example, let's define an `OptionParser` that resolves conflicts intelligently and add conflicting options to it :

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, `optparse` detects that a previously-added option is already using the `-n` option string. Since `conflict_handler` is "resolve", it resolves the situation by removing `-n` from the earlier option's list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that :

```
Options:
  --dry-run      do no harm
  ...
  -n, --noisy    be noisy
```

It's possible to whittle away the option strings for a previously-added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, `optparse` removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing `OptionParser` :

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n/--dry-run` option is no longer accessible, so `optparse` removes it, leaving this help text :

```
Options:
  ...
  -n, --noisy    be noisy
  --dry-run      new dry-run option
```

Nettoyage

`OptionParser` instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling `destroy()` on your `OptionParser` once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your `OptionParser`.

Other methods

`OptionParser` supports several other public methods :

`OptionParser.set_usage(usage)`

Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

`OptionParser.print_usage(file=None)`

Print the usage message for the current program (`self.usage`) to `file` (default `stdout`). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

`OptionParser.get_usage()`

Same as `print_usage()` but returns the usage string instead of printing it.

`OptionParser.set_defaults(dest=value, ...)`

Set default values for several option destinations at once. Using `set_defaults()` is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several « mode » options all set the same destination, any one of them can set the default, and the last one wins :

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")      # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")    # overrides above setting
```

To avoid this confusion, use `set_defaults()` :

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

36.1.4 Option Callbacks

When `optparse`'s built-in actions and types aren't quite enough for your needs, you have two choices : extend `optparse` or define a callback option. Extending `optparse` is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option :

- define the option itself using the "callback" action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the `OptionParser.add_option()` method. Apart from `action`, the only option attribute you must specify is `callback`, the function to call :

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, `optparse` doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments—the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

`optparse` always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via `callback_args` and `callback_kwargs`. Thus, the minimal callback function signature is :

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option :

`type` has its usual meaning : as with the "store" or "append" actions, it instructs `optparse` to consume one argument and convert it to `type`. Rather than storing the converted value(s) anywhere, though, `optparse` passes it to your callback function.

`nargs` also has its usual meaning : if it is supplied and `> 1`, `optparse` will consume `nargs` arguments, each of which must be convertible to `type`. It then passes a tuple of converted values to your callback.

`callback_args` a tuple of extra positional arguments to pass to the callback

`callback_kwargs` a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows :

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

`option` is the Option instance that's calling the callback

`opt_str` is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string—e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `--foobar`.)

`value` is the argument to this option seen on the command-line. `optparse` will only expect an argument if `type` is set; the type of `value` will be the type implied by the option's type. If `type` for this option is `None` (no argument expected), then `value` will be `None`. If `nargs > 1`, `value` will be a tuple of values of the appropriate type.

`parser` is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes :

`parser.largs` the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of `parse_args()`.)

`parser.rargs` the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

parser.values the object where option values are by default stored (an instance of `optparse.OptionValues`). This lets callbacks use the same mechanism as the rest of *optparse* for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args is a tuple of arbitrary positional arguments supplied via the *callback_args* option attribute.

kwargs is a dictionary of arbitrary keyword arguments supplied via *callback_kwargs*.

Raising errors in a callback

The callback function should raise `OptionValueError` if there are any problems with the option or its argument(s). *optparse* catches this and terminates the program, printing the error message you supply to `stderr`. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1 : trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen :

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the "store_true" action.

Callback example 2 : check option order

Here's a slightly more interesting example : record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3 : check option order (generalized)

If you want to re-use this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work : the error message and the flag that it sets must be generalized.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

Callback example 4 : check arbitrary condition

Of course, you could put any condition in there—you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this :

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                                % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5 : fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a "store" or "append" option : if you define `type`, then the option takes one argument that must be convertible to that type ; if you further define `nargs`, then the option takes `nargs` arguments.

Here's an example that just emulates the standard "store" action :

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that `optparse` takes care of consuming 3 arguments and converting them to integers for you ; all you have to do is store them. (Or whatever ; obviously you don't need a callback for this example.)

Callback example 6 : variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as `optparse` doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that `optparse` normally handles for you. In particular, callbacks should implement the conventional rules for bare `--` and `-` arguments :

- either `--` or `-` can be option arguments
- bare `--` (if not the argument to some option) : halt command-line processing and discard the `--`
- bare `-` (if not the argument to some option) : halt command-line processing but keep the `-` (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why `optparse` doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments :

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []
```

(suite sur la page suivante)

```

def floatable(str):
    try:
        float(str)
        return True
    except ValueError:
        return False

for arg in parser.rargs:
    # stop on --foo like options
    if arg[:2] == "--" and len(arg) > 2:
        break
    # stop on -a, but not on -3 or -3.0
    if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
        break
    value.append(arg)

del parser.rargs[:len(value)]
setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)

```

36.1.5 Extending optparse

Since the two major controlling factors in how *optparse* interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of *optparse*'s `Option` class. This class has a couple of attributes that define *optparse*'s types : `TYPES` and `TYPE_CHECKER`.

`Option.TYPES`

A tuple of type names ; in your subclass, simply define a new tuple `TYPES` that builds on the standard one.

`Option.TYPE_CHECKER`

A dictionary mapping type names to type-checking functions. A type-checking function has the following signature :

```
def check_mytype(option, opt, value)
```

where `option` is an `Option` instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the `OptionValues` instance returned by `OptionParser.parse_args()`, or be passed to a callback as the `value` parameter.

Your type-checking function should raise `OptionValueError` if it encounters any problems. `OptionValueError` takes a single string argument, which is passed as-is to *OptionParser*'s `error()` method, which in turn prepends the program name and the string `"error: "` and prints everything to `stderr` before terminating the process.

Here's a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because *optparse* 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports :

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the `TYPE_CHECKER` class attribute of your `Option` subclass) :

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the `Option` subclass :

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a `copy()` of `Option.TYPE_CHECKER`, we would end up modifying the `TYPE_CHECKER` attribute of `optparse`'s `Option` class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it ! Now you can write a script that uses the new option type just like any other `optparse`-based script, except you have to instruct your `OptionParser` to use `MyOption` instead of `Option` :

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to `OptionParser`; if you don't use `add_option()` in the above way, you don't need to tell `OptionParser` which option class to use :

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that `optparse` has a couple of classifications for actions :

- « **store** » **actions** actions that result in `optparse` storing a value to an attribute of the current `OptionValues` instance; these options require a `dest` attribute to be supplied to the `Option` constructor.

- « **typed** » **actions** actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a `type` attribute to the `Option` constructor.

These are overlapping sets : some default « **store** » actions are "store", "store_const", "append", and "count", while the default « **typed** » actions are "store", "append", and "callback".

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of `Option` (all are lists of strings) :

`Option.ACTIONS`

All actions must be listed in `ACTIONS`.

`Option.STORE_ACTIONS`

« **store** » actions are additionally listed here.

Option.TYPED_ACTIONS

« typed » actions are additionally listed here.

Option.ALWAYS_TYPED_ACTIONS

Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that *optparse* assigns the default type, "string", to options with no explicit type whose action is listed in *ALWAYS_TYPED_ACTIONS*.

In order to actually implement your new action, you must override Option's `take_action()` method and add a case that recognizes your action.

For example, let's add an "extend" action. This is similar to the standard "append" action, but instead of taking a single value from the command-line and appending it to an existing list, "extend" will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if `--names` is an "extend" option of type "string", the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option :

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note :

- "extend" both expects a value on the command-line and stores that value somewhere, so it goes in both *STORE_ACTIONS* and *TYPED_ACTIONS*.
- to ensure that *optparse* assigns the default type of "string" to "extend" actions, we put the "extend" action in *ALWAYS_TYPED_ACTIONS* as well.
- `MyOption.take_action()` implements just this one new action, and passes control back to `Option.take_action()` for the standard *optparse* actions.
- `values` is an instance of the `optparse_parser.Values` class, which provides the very useful `ensure_value()` method. `ensure_value()` is essentially `getattr()` with a safety valve; it is called as

```
values.ensure_value(attr, value)
```

If the `attr` attribute of `values` doesn't exist or is `None`, then `ensure_value()` first sets it to `value`, and then returns "value. This is very handy for actions like "extend", "append", and "count", all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using `ensure_value()` means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as `None` and `ensure_value()` will take care of getting it right when it's needed.

36.2 `imp` — Access the import internals

Source code : [Lib/imp.py](#)

Obsolète depuis la version 3.4 : The `imp` package is pending deprecation in favor of `importlib`.

This module provides an interface to the mechanisms used to implement the `import` statement. It defines the following constants and functions :

`imp.get_magic()`

Return the magic string value used to recognize byte-compiled code files (`.pyc` files). (This value may be different for each Python version.)

Obsolète depuis la version 3.4 : Use `importlib.util.MAGIC_NUMBER` instead.

`imp.get_suffixes()`

Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form `(suffix, mode, type)`, where `suffix` is a string to be appended to the module name to form the filename to search for, `mode` is the mode string to pass to the built-in `open()` function to open the file (this can be `'r'` for text files or `'rb'` for binary files), and `type` is the file type, which has one of the values `PY_SOURCE`, `PY_COMPILED`, or `C_EXTENSION`, described below.

Obsolète depuis la version 3.3 : Use the constants defined on `importlib.machinery` instead.

`imp.find_module(name[, path])`

Try to find the module `name`. If `path` is omitted or `None`, the list of directory names given by `sys.path` is searched, but first a few special places are searched : the function tries to find a built-in module with the given name (`C_BUILTIN`), then a frozen module (`PY_FROZEN`), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

Otherwise, `path` must be a list of directory names; each directory is searched for files with any of the suffixes returned by `get_suffixes()` above. Invalid names in the list are silently ignored (but all list items must be strings).

If search is successful, the return value is a 3-element tuple `(file, pathname, description)` :

`file` is an open *file object* positioned at the beginning, `pathname` is the pathname of the file found, and `description` is a 3-element tuple as contained in the list returned by `get_suffixes()` describing the kind of module found.

If the module does not live in a file, the returned `file` is `None`, `pathname` is the empty string, and the `description` tuple contains empty strings for its suffix and mode; the module type is indicated as given in parentheses above. If the search is unsuccessful, `ImportError` is raised. Other exceptions indicate problems with the arguments or environment.

If the module is a package, `file` is `None`, `pathname` is the package path and the last item in the `description` tuple is `PKG_DIRECTORY`.

This function does not handle hierarchical module names (names containing dots). In order to find `P.M`, that is, submodule `M` of package `P`, use `find_module()` and `load_module()` to find and load package `P`, and then use `find_module()` with the `path` argument set to `P.__path__`. When `P` itself has a dotted name, apply this recipe recursively.

Obsolète depuis la version 3.3 : Use `importlib.util.find_spec()` instead unless Python 3.3 compatibility is required, in which case use `importlib.find_loader()`. For example usage of the former case, see the *Examples* section of the `importlib` documentation.

`imp.load_module(name, file, pathname, description)`

Load a module that was previously found by `find_module()` (or by an otherwise conducted search yielding compatible results). This function does more than importing the module : if the module was already imported, it will reload the module ! The `name` argument indicates the full module name (including the package name, if this is a submodule of a package). The `file` argument is an open file, and `pathname` is the corresponding file name; these can be `None` and `' '`, respectively, when the module is a package or not being loaded from a file. The `description` argument is a tuple, as would be returned by `get_suffixes()`, describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually `ImportError`) is raised.

Important : the caller is responsible for closing the *file* argument, if it was not `None`, even when an exception is raised. This is best done using a `try ... finally` statement.

Obsolète depuis la version 3.3 : If previously used in conjunction with `imp.find_module()` then consider using `importlib.import_module()`, otherwise use the loader returned by the replacement you chose for `imp.find_module()`. If you called `imp.load_module()` and related functions directly with file path arguments then use a combination of `importlib.util.spec_from_file_location()` and `importlib.util.module_from_spec()`. See the *Examples* section of the `importlib` documentation for details of the various approaches.

`imp.new_module(name)`

Return a new empty module object called *name*. This object is *not* inserted in `sys.modules`.

Obsolète depuis la version 3.4 : Use `importlib.util.module_from_spec()` instead.

`imp.reload(module)`

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the *module* argument).

When `reload(module)` is executed :

- Python modules” code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module’s dictionary. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats :

When a module is reloaded, its dictionary (containing the module’s global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module’s advantage if it maintains a global table or cache of objects — with a `try` statement it can test for the table’s presence and skip its initialization if desired :

```
try:
    cache
except NameError:
    cache = {}
```

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for `sys`, `__main__` and `builtins`. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from ... import ...`, calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (`module.*name*`) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

Modifié dans la version 3.3 : Relies on both `__name__` and `__loader__` being defined on the module being reloaded instead of just `__name__`.

Obsolète depuis la version 3.4 : Use `importlib.reload()` instead.

The following functions are conveniences for handling **PEP 3147** byte-compiled file paths.

Nouveau dans la version 3.2.

`imp.cache_from_source(path, debug_override=None)`

Return the [PEP 3147](#) path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see `get_tag()`; if `sys.implementation.cache_tag` is not defined then `NotImplementedError` will be raised). By passing in `True` or `False` for *debug_override* you can override the system's value for `__debug__`, leading to optimized bytecode.

path need not exist.

Modifié dans la version 3.3 : If `sys.implementation.cache_tag` is `None`, then `NotImplementedError` is raised.

Obsolète depuis la version 3.4 : Use `importlib.util.cache_from_source()` instead.

Modifié dans la version 3.5 : The *debug_override* parameter no longer creates a `.pyo` file.

`imp.source_from_cache(path)`

Given the *path* to a [PEP 3147](#) file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to [PEP 3147](#) format, a `ValueError` is raised. If `sys.implementation.cache_tag` is not defined, `NotImplementedError` is raised.

Modifié dans la version 3.3 : Raise `NotImplementedError` when `sys.implementation.cache_tag` is not defined.

Obsolète depuis la version 3.4 : Use `importlib.util.source_from_cache()` instead.

`imp.get_tag()`

Return the [PEP 3147](#) magic tag string matching this version of Python's magic number, as returned by `get_magic()`.

Obsolète depuis la version 3.4 : Use `sys.implementation.cache_tag` directly starting in Python 3.3.

The following functions help interact with the import system's internal locking mechanism. Locking semantics of imports are an implementation detail which may vary from release to release. However, Python ensures that circular imports work without any deadlocks.

`imp.lock_held()`

Return `True` if the global import lock is currently held, else `False`. On platforms without threads, always return `False`.

On platforms with threads, a thread executing an import first holds a global import lock, then sets up a per-module lock for the rest of the import. This blocks other threads from importing the same module until the original import completes, preventing other threads from seeing incomplete module objects constructed by the original thread. An exception is made for circular imports, which by construction have to expose an incomplete module object at some point.

Modifié dans la version 3.3 : The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

Obsolète depuis la version 3.4.

`imp.acquire_lock()`

Acquire the interpreter's global import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules.

Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

Modifié dans la version 3.3 : The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

Obsolète depuis la version 3.4.

`imp.release_lock()`

Release the interpreter's global import lock. On platforms without threads, this function does nothing.

Modifié dans la version 3.3 : The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

Obsolète depuis la version 3.4.

The following constants with integer values, defined in this module, are used to indicate the search result of `find_module()`.

`imp.PY_SOURCE`

The module was found as a source file.

Obsolète depuis la version 3.3.

`imp.PY_COMPILED`

The module was found as a compiled code object file.

Obsolète depuis la version 3.3.

`imp.C_EXTENSION`

The module was found as dynamically loadable shared library.

Obsolète depuis la version 3.3.

`imp.PKG_DIRECTORY`

The module was found as a package directory.

Obsolète depuis la version 3.3.

`imp.C_BUILTIN`

The module was found as a built-in module.

Obsolète depuis la version 3.3.

`imp.PY_FROZEN`

The module was found as a frozen module.

Obsolète depuis la version 3.3.

class `imp.NullImporter` (*path_string*)

The `NullImporter` type is a **PEP 302** import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises `ImportError`. Otherwise, a `NullImporter` instance is returned.

Instances have only one method :

find_module (*fullname* [, *path*])

This method always returns `None`, indicating that the requested module could not be found.

Modifié dans la version 3.3 : `None` is inserted into `sys.path_importer_cache` instead of an instance of `NullImporter`.

Obsolète depuis la version 3.4 : Insert `None` into `sys.path_importer_cache` instead.

36.2.1 Exemples

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This *implementation* wouldn't work in that version, since `find_module()` has been extended and `load_module()` has been added in 1.4.)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
```

(suite sur la page suivante)

(suite de la page précédente)

```
except KeyError:
    pass

# If any of the following calls raises an exception,
# there's a problem we can't handle -- let the caller handle it.

fp, pathname, description = imp.find_module(name)

try:
    return imp.load_module(name, fp, pathname, description)
finally:
    # Since we may exit via an exception, close fp explicitly.
    if fp:
        fp.close()
```

Modules non Documentés

Voici un rapide survol des modules actuellement non documentés, mais qui devraient l'être. N'hésitez pas à contribuer à leur documentation ! (En anglais, par mail, à docs@python.org.)

L'idée de lister les modules à documenter vient d'un message de Fredrik Lundh. Depuis, le contenu de ce chapitre a pu être mis à jour.

37.1 Modules spécifiques à une plateforme

Ces modules sont utilisés pour implémenter le module `os.path`, et n'ont pas plus de documentation. Le besoin de les documenter ne se fait pas réellement sentir.

ntpath — Implémentation de `os.path` pour les plateformes Win32 et Win64.

posixpath — Implémentation du module `os.path` pour les systèmes POSIX.

>>> L'invite de commande utilisée par défaut dans l'interpréteur interactif. On la voit souvent dans des exemples de code qui peuvent être exécutés interactivement dans l'interpréteur.

. . . L'invite de commande utilisée par défaut dans l'interpréteur interactif lorsqu'on entre un bloc de code indenté, dans des délimiteurs fonctionnant par paires (parenthèses, crochets, accolades, triple guillemets), ou après un décorateur.

2to3 Outil qui essaie de convertir du code pour Python 2.x en code pour Python 3.x en gérant la plupart des incompatibilités qui peuvent être détectées en analysant la source et parcourant son arbre syntaxique.

`2to3` est disponible dans la bibliothèque standard sous le nom de `lib2to3`; un point d'entrée indépendant est fourni via `Tools/scripts/2to3`. Cf. [2to3](#) — *Traduction automatique de code en Python 2 vers Python 3*.

classe de base abstraite Les classes de base abstraites (ABC, suivant l'abréviation anglaise *Abstract Base Class*) complètent le *duck-typing* en fournissant un moyen de définir des interfaces pour les cas où d'autres techniques comme `hasattr()` seraient inélégantes ou subtilement fausses (par exemple avec les méthodes magiques). Les ABC introduisent des sous-classes virtuelles qui n'héritent pas d'une classe mais qui sont quand même reconnues par `isinstance()` ou `issubclass()` (voir la documentation du module `abc`). Python contient de nombreuses ABC pour les structures de données (dans le module `collections.abc`), les nombres (dans le module `numbers`), les flux (dans le module `io`) et les chercheurs-chargeurs du système d'importation (dans le module `importlib.abc`). Vous pouvez créer vos propres ABC avec le module `abc`.

annotation Étiquette associée à une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour. Elle est utilisée par convention comme *type hint*.

Les annotations de variables locales ne sont pas accessibles au moment de l'exécution, mais les annotations de variables globales, d'attributs de classe et de fonctions sont stockées dans l'attribut spécial `__annotations__` des modules, classes et fonctions, respectivement.

Voir *variable annotation*, *function annotation*, **PEP 484** et **PEP 526**, qui décrivent cette fonctionnalité.

argument Valeur, donnée à une *fonction* ou à une *méthode* lors de son appel. Il existe deux types d'arguments :

— *argument nommé* : un argument précédé d'un identifiant (comme `name=`) ou un dictionnaire précédé de `**`, lors d'un appel de fonction. Par exemple, 3 et 5 sont tous les deux des arguments nommés dans l'appel à `complex()` ici :

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argument positionnel* : un argument qui n'est pas nommé. Les arguments positionnels apparaissent au début de la liste des arguments, ou donnés sous forme d'un *itérable* précédé par `*`. Par exemple, 3 et 5 sont tous les deux des arguments positionnels dans les appels suivants :

```
complex(3, 5)
complex(*(3, 5))
```

Les arguments se retrouvent dans le corps de la fonction appelée parmi les variables locales. Voir la section `calls` à propos des règles dictant cette affectation. Syntaxiquement, toute expression est acceptée comme argument, et c'est la valeur résultante de l'expression qui sera affectée à la variable locale.

Voir aussi *parameter* dans le glossaire, la question Différence entre argument et paramètre de la FAQ et la [PEP 362](#).

gestionnaire de contexte asynchrone (*asynchronous context manager* en anglais) Objet contrôlant l'environnement à l'intérieur d'une instruction `with` en définissant les méthodes `__aenter__()` et `__aexit__()`. A été introduit par la [PEP 492](#).

générateur asynchrone Fonction qui renvoie un *asynchronous generator iterator*. Cela ressemble à une coroutine définie par `async def`, sauf qu'elle contient une ou des expressions `yield` produisant ainsi une série de valeurs utilisables dans une boucle `async for`.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

Un générateur asynchrone peut contenir des expressions `await` ainsi que des instructions `async for`, et `async with`.

itérateur de générateur asynchrone Objet créé par une fonction *asynchronous generator*.

This is an *asynchronous iterator* which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Chaque `yield` suspend temporairement l'exécution, en gardant en mémoire l'endroit et l'état de l'exécution (ce qui inclut les variables locales et les `try` en cours). Lorsque l'exécution de l'itérateur de générateur asynchrone reprend avec un nouvel *awaitable* renvoyé par `__anext__()`, elle repart de là où elle s'était arrêtée. Voir la [PEP 492](#) et la [PEP 525](#).

itérable asynchrone Objet qui peut être utilisé dans une instruction `async for`. Sa méthode `__aiter__()` doit renvoyer un *asynchronous iterator*. A été introduit par la [PEP 492](#).

itérateur asynchrone An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__()` must return an *awaitable* object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a *StopAsyncIteration* exception. Introduced by [PEP 492](#).

attribut Valeur associée à un objet et désignée par son nom via une notation utilisant des points. Par exemple, si un objet *o* possède un attribut *a*, il sera référencé par *o.a*.

awaitable Objet pouvant être utilisé dans une expression `await`. Ce peut être une *coroutine* ou un objet avec une méthode `__await__()`. Voir aussi la [PEP 492](#).

BDFL Dictateur bienveillant à vie (*Benevolent Dictator For Life* en anglais). Pseudonyme de Guido van Rossum, le créateur de Python.

fichier binaire Un *file object* capable de lire et d'écrire des *bytes-like objects*. Des fichiers binaires sont, par exemple, les fichiers ouverts en mode binaire ('rb', 'wb', ou 'rb+'), `sys.stdin.buffer`, `sys.stdout.buffer`, les instances de `io.BytesIO` ou de `gzip.GzipFile`.

Consultez *fichier texte*, un objet fichier capable de lire et d'écrire des objets *str*.

objet octet-compatible Un objet gérant les *bufferobjects* et pouvant exporter un tampon (*buffer* en anglais) *C-contiguous*. Cela inclut les objets *bytes*, *bytearray* et *array.array*, ainsi que beaucoup d'objets *memoryview*. Les objets *bytes-compatible* peuvent être utilisés pour diverses opérations sur des données binaires, comme la compression, la sauvegarde dans un fichier binaire ou l'envoi sur le réseau.

Certaines opérations nécessitent de travailler sur des données binaires variables. La documentation parle de ceux-ci comme des *read-write bytes-like objects*. Par exemple, *bytearray* ou une *memoryview* d'un *bytearray* en font partie. D'autres opérations nécessitent de travailler sur des données binaires stockées dans des objets immuables (« *read-only bytes-like objects* »), par exemples *bytes* ou *memoryview* d'un objet *byte*.

code intermédiaire (*bytecode*) Le code source, en Python, est compilé en un code intermédiaire (*bytecode* en anglais), la représentation interne à CPython d'un programme Python. Le code intermédiaire est mis en cache dans un fichier `.pyc` de manière à ce qu'une seconde exécution soit plus rapide (la compilation en code intermédiaire a déjà été faite). On dit que ce *langage intermédiaire* est exécuté sur une *virtual machine* qui exécute des instructions machine pour chaque instruction du code intermédiaire. Notez que le code intermédiaire n'a pas vocation à fonctionner sur différentes machines virtuelles Python ou à être stable entre différentes versions de Python.

La documentation du *module* `dis` fournit une liste des instructions du code intermédiaire.

classe Modèle pour créer des objets définis par l'utilisateur. Une définition de classe (*class*) contient normalement des définitions de méthodes qui agissent sur les instances de la classe.

variable de classe Une variable définie dans une classe et destinée à être modifiée uniquement au niveau de la classe (c'est-à-dire, pas dans une instance de la classe).

coercition Conversion implicite d'une instance d'un type vers un autre lors d'une opération dont les deux opérandes doivent être de même type. Par exemple `int(3.15)` convertit explicitement le nombre à virgule flottante en nombre entier 3. Mais dans l'opération `3 + 4.5`, les deux opérandes sont d'un type différent, alors qu'elles doivent avoir le même type pour être additionnées (sinon une exception `TypeError` serait levée). Sans coercition, toutes les opérandes, même de types compatibles, devraient être converties (on parle aussi de *cast*) explicitement par le développeur, par exemple : `float(3) + 4.5` au lieu du simple `3 + 4.5`.

nombre complexe Extension des nombres réels familiers, dans laquelle tous les nombres sont exprimés sous la forme d'une somme d'une partie réelle et d'une partie imaginaire. Les nombres imaginaires sont les nombres réels multipliés par l'unité imaginaire (la racine carrée de -1 , souvent écrite *i* en mathématiques ou *j* par les ingénieurs). Python comprend nativement les nombres complexes, écrits avec cette dernière notation : la partie imaginaire est écrite avec un suffixe *j*, exemple, `3+1j`. Pour utiliser les équivalents complexes de *math*, utilisez *cmath*. Les nombres complexes sont un concept assez avancé en mathématiques. Si vous ne connaissez pas ce concept, vous pouvez tranquillement les ignorer.

gestionnaire de contexte Objet contrôlant l'environnement à l'intérieur d'un bloc `with` en définissant les méthodes `__enter__()` et `__exit__()`. Consultez la [PEP 343](#).

contigu Un tampon (*buffer* en anglais) est considéré comme contigu s'il est soit *C-contigu* soit *Fortran-contigu*. Les tampons de dimension zéro sont C-contigus et Fortran-contigus. Pour un tableau à une dimension, ses éléments doivent être placés en mémoire l'un à côté de l'autre, dans l'ordre croissant de leur indice, en commençant à zéro. Pour qu'un tableau multidimensionnel soit C-contigu, le dernier indice doit être celui qui varie le plus rapidement lors du parcours de ses éléments dans l'ordre de leur adresse mémoire. À l'inverse, dans les tableaux Fortran-contigu, c'est le premier indice qui doit varier le plus rapidement.

coroutine Les coroutines sont une forme généralisées des fonctions. On entre dans une fonction en un point et on en sort en un autre point. On peut entrer, sortir et reprendre l'exécution d'une coroutine en plusieurs points. Elles peuvent être implémentées en utilisant l'instruction `async def`. Voir aussi la [PEP 492](#).

fonction coroutine Fonction qui renvoie un objet *coroutine*. Une fonction coroutine peut être définie par l'instruction `async def` et peut contenir les mots clés `await`, `async for` ainsi que `async with`. A été introduit par la [PEP 492](#).

CPython L'implémentation canonique du langage de programmation Python, tel que distribué sur [python.org](#). Le terme « CPython » est utilisé dans certains contextes lorsqu'il est nécessaire de distinguer cette implémentation des autres comme *Jython* ou *IronPython*.

décorateur Fonction dont la valeur de retour est une autre fonction. Un décorateur est habituellement utilisé pour transformer une fonction via la syntaxe `@wrapper`, dont les exemples typiques sont : `classmethod()` et `staticmethod()`.

La syntaxe des décorateurs est simplement du sucre syntaxique, les définitions des deux fonctions suivantes sont sémantiquement équivalentes :

```
def f(...):
    ...
f = staticmethod(f)
```

(suite sur la page suivante)

```
@staticmethod
def f(...):
    ...
```

Quoique moins fréquemment utilisé, le même concept existe pour les classes. Consultez la documentation définitions de fonctions et définitions de classes pour en savoir plus sur les décorateurs.

descripteur N'importe quel objet définissant les méthodes `__get__()`, `__set__()`, ou `__delete__()`. Lorsque l'attribut d'une classe est un descripteur, son comportement spécial est déclenché lors de la recherche des attributs. Normalement, lorsque vous écrivez `a.b` pour obtenir, affecter ou effacer un attribut, Python recherche l'objet nommé `b` dans le dictionnaire de la classe de `a`. Mais si `b` est un descripteur, c'est la méthode de ce descripteur qui est alors appelée. Comprendre les descripteurs est requis pour avoir une compréhension approfondie de Python, ils sont la base de nombre de ses caractéristiques notamment les fonctions, méthodes, propriétés, méthodes de classes, méthodes statiques et les références aux classes parentes.

Pour plus d'informations sur les méthodes des descripteurs, consultez `descriptors`.

dictionnaire Structure de donnée associant des clés à des valeurs. Les clés peuvent être n'importe quel objet possédant les méthodes `__hash__()` et `__eq__()`. En Perl, les dictionnaires sont appelés « *hash* ».

vue de dictionnaire Objets retournés par les méthodes `dict.keys()`, `dict.values()` et `dict.items()`. Ils fournissent des vues dynamiques des entrées du dictionnaire, ce qui signifie que lorsque le dictionnaire change, la vue change. Pour transformer une vue en vraie liste, utilisez `list(dictview)`. Voir [Les vues de dictionnaires](#).

docstring (chaîne de documentation) Première chaîne littérale qui apparaît dans l'expression d'une classe, fonction, ou module. Bien qu'ignorée à l'exécution, elle est reconnue par le compilateur et placée dans l'attribut `__doc__` de la classe, de la fonction ou du module. Comme cette chaîne est disponible par introspection, c'est l'endroit idéal pour documenter l'objet.

duck-typing Style de programmation qui ne prend pas en compte le type d'un objet pour déterminer s'il respecte une interface, mais qui appelle simplement la méthode ou l'attribut (Si ça a un bec et que ça cancanne, ça doit être un canard, *duck* signifie canard en anglais). En se concentrant sur les interfaces plutôt que les types, du code bien construit améliore sa flexibilité en autorisant des substitutions polymorphiques. Le *duck-typing* évite de vérifier les types via `type()` ou `isinstance()`, Notez cependant que le *duck-typing* peut travailler de pair avec les *classes de base abstraites*. À la place, le *duck-typing* utilise plutôt `hasattr()` ou la programmation *EAFP*.

EAFP Il est plus simple de demander pardon que demander la permission (*Easier to Ask for Forgiveness than Permission* en anglais). Ce style de développement Python fait l'hypothèse que le code est valide et traite les exceptions si cette hypothèse s'avère fausse. Ce style, propre et efficace, est caractérisé par la présence de beaucoup de mots clés `try` et `except`. Cette technique de programmation contraste avec le style *LYBL* utilisé couramment dans les langages tels que C.

expression Suite logique de termes et chiffres conformes à la syntaxe Python dont l'évaluation fournit une valeur. En d'autres termes, une expression est une suite d'éléments tels que des noms, opérateurs, littéraux, accès d'attributs, méthodes ou fonctions qui aboutissent à une valeur. Contrairement à beaucoup d'autres langages, les différentes constructions du langage ne sont pas toutes des expressions. On trouve également des *instructions* qui ne peuvent pas être utilisées comme expressions, tel que `if`. Les affectations sont également des instructions et non des expressions.

module d'extension Module écrit en C ou C++, utilisant l'API C de Python pour interagir avec Python et le code de l'utilisateur.

f-string Chaîne littérale préfixée de 'f' ou 'F'. Les « f-strings » sont un raccourci pour formatted string literals. Voir la [PEP 498](#).

objet fichier Objet exposant une ressource via une API orientée fichier (avec les méthodes `read()` ou `write()`). En fonction de la manière dont il a été créé, un objet fichier peut interfacer l'accès à un fichier sur le disque ou à un autre type de stockage ou de communication (typiquement l'entrée standard, la sortie standard, un tampon en mémoire, un connecteur réseau...). Les objets fichiers sont aussi appelés *file-like-objects* ou *streams*.

Il existe en réalité trois catégories de fichiers objets : les *fichiers binaires* bruts, les *fichiers binaires* avec tampon (*buffer*) et les *fichiers textes*. Leurs interfaces sont définies dans le module `io`. Le moyen le plus simple et direct de créer un objet fichier est d'utiliser la fonction `open()`.

objet fichier-compatible Synonyme de *objet fichier*.

chercheur Objet qui essaie de trouver un *chargeur* pour le module en cours d'importation.

Depuis Python 3.3, il existe deux types de chercheurs : les *chercheurs dans les méta-chemins* à utiliser avec `sys.meta_path`; les *chercheurs d'entrée dans path* à utiliser avec `sys.path_hooks`.

Voir les [PEP 302](#), [PEP 420](#) et [PEP 451](#) pour plus de détails.

division entière Division mathématique arrondissant à l'entier inférieur. L'opérateur de la division entière est `//`. Par exemple l'expression `11 // 4` vaut 2, contrairement à `11 / 4` qui vaut 2.75. Notez que `(-11) // 4` vaut -3 car l'arrondi se fait à l'entier inférieur. Voir la [PEP 328](#).

fonction Suite d'instructions qui renvoie une valeur à son appelant. On peut lui passer des *arguments* qui pourront être utilisés dans le corps de la fonction. Voir aussi *paramètre*, *méthode* et *fonction*.

annotation de fonction *annotation* d'un paramètre de fonction ou valeur de retour.

Function annotations are usually used for *type hints* : for example, this function is expected to take two *int* arguments and is also expected to have an *int* return value :

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

L'annotation syntaxique de la fonction est expliquée dans la section *fonction*.

Voir *variable annotation* et [PEP 484](#), qui décrivent cette fonctionnalité.

__future__ Pseudo-module que les développeurs peuvent utiliser pour activer de nouvelles fonctionnalités du langage qui ne sont pas compatibles avec l'interpréteur utilisé.

En important le module `__future__` et en affichant ses variables, vous pouvez voir à quel moment une nouvelle fonctionnalité a été rajoutée dans le langage et quand elle devient le comportement par défaut :

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

ramasse-miettes (*garbage collection* en anglais) Mécanisme permettant de libérer de la mémoire lorsqu'elle n'est plus utilisée. Python utilise un ramasse-miettes par comptage de référence et un ramasse-miettes cyclique capable de détecter et casser les références circulaires. Le ramasse-miettes peut être contrôlé en utilisant le module `gc`.

générateur Fonction qui renvoie un *itérateur de générateur*. Cela ressemble à une fonction normale, en dehors du fait qu'elle contient une ou des expressions `yield` produisant une série de valeurs utilisable dans une boucle `for` ou récupérées une à une via la fonction `next()`.

Fait généralement référence à une fonction générateur mais peut faire référence à un *itérateur de générateur* dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser les termes complets lève l'ambiguïté.

itérateur de générateur Objet créé par une fonction *générateur*.

Chaque `yield` suspend temporairement l'exécution, en se rappelant l'endroit et l'état de l'exécution (y compris les variables locales et les `try` en cours). Lorsque l'itérateur de générateur reprend, il repart là où il en était (contrairement à une fonction qui prendrait un nouveau départ à chaque invocation).

expression génératrice Expression qui donne un itérateur. Elle ressemble à une expression normale, suivie d'une expression `for` définissant une variable de boucle, un intervalle et une expression `if` optionnelle. Toute cette expression génère des valeurs pour la fonction qui l'entoure :

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

fonction générique Fonction composée de plusieurs fonctions implémentant les mêmes opérations pour différents types. L'implémentation à utiliser est déterminée lors de l'appel par l'algorithme de répartition.

Voir aussi *single dispatch*, le décorateur `functools singledispatch()` et la [PEP 443](#).

GIL Voir *global interpreter lock*.

verrou global de l'interpréteur (*global interpreter lock* en anglais) Mécanisme utilisé par l'interpréteur *CPython* pour s'assurer qu'un seul fil d'exécution (*thread* en anglais) n'exécute le *bytecode* à la fois. Cela simplifie l'implémentation de CPython en rendant le modèle objet (incluant des parties critiques comme la classe native *dict*) implicitement protégé contre les accès concourants. Verrouiller l'interpréteur entier rend plus facile l'implémentation de multiples fils d'exécution (*multi-thread* en anglais), au détriment malheureusement de beaucoup du parallélisme possible sur les machines ayant plusieurs processeurs.

Cependant, certains modules d'extension, standards ou non, sont conçus de manière à libérer le GIL lorsqu'ils effectuent des tâches lourdes tel que la compression ou le hachage. De la même manière, le GIL est toujours libéré lors des entrées / sorties.

Les tentatives précédentes d'implémenter un interpréteur Python avec une granularité de verrouillage plus fine ont toutes échouées, à cause de leurs mauvaises performances dans le cas d'un processeur unique. Il est admis que corriger ce problème de performance induit mènerait à une implémentation beaucoup plus compliquée et donc plus coûteuse à maintenir.

hachable Un objet est *hachable* s'il a une empreinte (*hash*) qui ne change jamais (il doit donc implémenter une méthode `__hash__()`) et s'il peut être comparé à d'autres objets (avec la méthode `__eq__()`). Les objets hachables dont la comparaison par `__eq__` est vraie doivent avoir la même empreinte.

La hachabilité permet à un objet d'être utilisé comme clé de dictionnaire ou en tant que membre d'un ensemble (type *set*), car ces structures de données utilisent ce *hash*.

Tous les types immuables natifs de Python sont hachables, mais les conteneurs muables (comme les listes ou les dictionnaires) ne le sont pas. Toutes les instances de classes définies par les utilisateurs sont hachables par défaut. Elles sont toutes considérées différentes (sauf avec elles-mêmes) et leur valeur de hachage est calculée à partir de leur `id()`.

IDLE Environnement de développement intégré pour Python. IDLE est un éditeur basique et un interpréteur livré avec la distribution standard de Python.

immuable Objet dont la valeur ne change pas. Les nombres, les chaînes et les n-uplets sont immuables. Ils ne peuvent être modifiés. Un nouvel objet doit être créé si une valeur différente doit être stockée. Ils jouent un rôle important quand une valeur de *hash* constante est requise, typiquement en clé de dictionnaire.

chemin des importations Liste de *entrées* dans lesquelles le *chercheur basé sur les chemins* cherche les modules à importer. Typiquement, lors d'une importation, cette liste vient de `sys.path`; pour les sous-paquets, elle peut aussi venir de l'attribut `__path__` du paquet parent.

importer Processus rendant le code Python d'un module disponible dans un autre.

importateur Objet qui trouve et charge un module, en même temps un *chercheur* et un *chargeur*.

interactif Python a un interpréteur interactif, ce qui signifie que vous pouvez écrire des expressions et des instructions à l'invite de l'interpréteur. L'interpréteur Python va les exécuter immédiatement et vous en présenter le résultat. Démarrez juste `python` (probablement depuis le menu principal de votre ordinateur). C'est un moyen puissant pour tester de nouvelles idées ou étudier de nouveaux modules (souvenez-vous de `help(x)`).

interprété Python est un langage interprété, en opposition aux langages compilés, bien que la frontière soit floue en raison de la présence d'un compilateur en code intermédiaire. Cela signifie que les fichiers sources peuvent être exécutés directement, sans avoir à compiler un fichier exécutable intermédiaire. Les langages interprétés ont généralement un cycle de développement / débogage plus court que les langages compilés. Cependant, ils s'exécutent généralement plus lentement. Voir aussi *interactif*.

arrêt de l'interpréteur Lorsqu'on lui demande de s'arrêter, l'interpréteur Python entre dans une phase spéciale où il libère graduellement les ressources allouées, comme les modules ou quelques structures de données internes. Il fait aussi quelques appels au *ramasse-miettes*. Cela peut déclencher l'exécution de code dans des destructeurs ou des fonctions de rappels de *weakrefs*. Le code exécuté lors de l'arrêt peut rencontrer des exceptions puisque les ressources auxquelles il fait appel sont susceptibles de ne plus fonctionner, (typiquement les modules des bibliothèques ou le mécanisme de *warning*).

La principale raison d'arrêt de l'interpréteur est que le module `__main__` ou le script en cours d'exécution a terminé de s'exécuter.

itérable Objet capable de renvoyer ses éléments un à un. Par exemple, tous les types séquence (comme *list*, *str*, et *tuple*), quelques autres types comme *dict*, *objets fichiers* ou tout objet d'une classe ayant une méthode `__iter__()` ou `__getitem__()` qui implémente la sémantique d'une *Sequence*.

Les itérables peuvent être utilisés dans des boucles `for` et à beaucoup d'autres endroits où une séquence est requise (`zip()`, `map()`...). Lorsqu'un itérable est passé comme argument à la fonction native `iter()`, celle-ci fournit en retour un itérateur sur cet itérable. Cet itérateur n'est valable que pour une seule passe sur le jeu de valeurs. Lors de l'utilisation d'itérables, il n'est habituellement pas nécessaire d'appeler `iter()` ou de s'occuper soi-même des objets itérateurs. L'instruction `for` le fait automatiquement pour vous, créant une variable temporaire anonyme pour garder l'itérateur durant la boucle. Voir aussi *itérateur*, *séquence* et *générateur*.

itérateur Objet représentant un flux de donnée. Des appels successifs à la méthode `__next__()` de l'itérateur (ou le passer à la fonction native `next()`) donne successivement les objets du flux. Lorsque plus aucune donnée n'est disponible, une exception `StopIteration` est levée. À ce point, l'itérateur est épuisé et tous les appels suivants à sa méthode `__next__()` lèveront encore une exception `StopIteration`. Les itérateurs doivent avoir une méthode `__iter__()` qui renvoie l'objet itérateur lui-même, de façon à ce que chaque itérateur soit aussi itérable et puisse être utilisé dans la plupart des endroits où d'autres itérables sont attendus. Une exception notable est un code qui tente plusieurs itérations complètes. Un objet conteneur, (tel que `list`) produit un nouvel itérateur neuf à chaque fois qu'il est passé à la fonction `iter()` ou s'il est utilisé dans une boucle `for`. Faire ceci sur un itérateur donnerait simplement le même objet itérateur épuisé utilisé dans son itération précédente, le faisant ressembler à un conteneur vide.

Vous trouverez davantage d'informations dans *Les types itérateurs*.

fonction clé Une fonction clé est un objet callable qui renvoie une valeur à fins de tri ou de classement. Par exemple, la fonction `locale.strxfrm()` est utilisée pour générer une clé de classement prenant en compte les conventions de classement spécifiques aux paramètres régionaux courants.

Plusieurs outils dans Python acceptent des fonctions clés pour déterminer comment les éléments sont classés ou groupés. On peut citer les fonctions `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` et `itertools.groupby()`.

Il existe plusieurs moyens de créer une fonction clé. Par exemple, la méthode `str.lower()` peut servir de fonction clé pour effectuer des recherches insensibles à la casse. Aussi, il est possible de créer des fonctions clés avec des expressions lambda, comme `lambda r: (r[0], r[2])`. Vous noterez que le module `operator` propose des constructeurs de fonctions clefs : `attrgetter()`, `itemgetter()` et `methodcaller()`. Voir *Comment Trier* pour des exemples de création et d'utilisation de fonctions clefs.

argument nommé Voir *argument*.

lambda Fonction anonyme sous la forme d'une *expression* et ne contenant qu'une seule expression, exécutée lorsque la fonction est appelée. La syntaxe pour créer des fonctions lambda est : `lambda [parameters]: expression`

LBYL Regarde avant de sauter, (*Look before you leap* en anglais). Ce style de programmation consiste à vérifier des conditions avant d'effectuer des appels ou des accès. Ce style contraste avec le style *EAFP* et se caractérise par la présence de beaucoup d'instructions `if`.

Dans un environnement avec plusieurs fils d'exécution (*multi-threaded* en anglais), le style *LBYL* peut engendrer un séquençement critique (*race condition* en anglais) entre le « regarde » et le « sauter ». Par exemple, le code `if key in mapping: return mapping[key]` peut échouer si un autre fil d'exécution supprime la clé `key` du `mapping` après le test mais avant l'accès. Ce problème peut être résolu avec des verrous (*locks*) ou avec l'approche *EAFP*.

list Un type natif de *sequence* dans Python. En dépit de son nom, une `list` ressemble plus à un tableau (*array* dans la plupart des langages) qu'à une liste chaînée puisque les accès se font en $O(1)$.

liste en compréhension (ou liste en intension) Écriture concise pour manipuler tout ou partie des éléments d'une séquence et renvoyer une liste contenant les résultats. `result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]` génère la liste composée des nombres pairs de 0 à 255 écrits sous formes de chaînes de caractères et en hexadécimal (0x...). La clause `if` est optionnelle. Si elle est omise, tous les éléments du `range(256)` seront utilisés.

chargeur Objet qui charge un module. Il doit définir une méthode nommée `load_module()`. Un chargeur est typiquement donné par un *chercheur*. Voir la **PEP 302** pour plus de détails et `importlib.ABC.Loader` pour sa *classe de base abstraite*.

tableau de correspondances (*mapping* en anglais) Conteneur permettant de rechercher des éléments à partir de clés et implémentant les méthodes spécifiées dans les *classes de base abstraites* `collections.abc.Mapping`

ou `collections.abc.MutableMapping`. Les classes suivantes sont des exemples de tableaux de correspondances : `dict`, `collections.defaultdict`, `collections.OrderedDict` et `collections.Counter`.

chercheur dans les méta-chemins Un *chercheur* renvoyé par une recherche dans `sys.meta_path`. Les chercheurs dans les méta-chemins ressemblent, mais sont différents des *chercheurs d'entrée dans path*.

Voir `importlib.abc.MetaPathFinder` pour les méthodes que les chercheurs dans les méta-chemins doivent implémenter.

métaclasse Classe d'une classe. Les définitions de classe créent un nom pour la classe, un dictionnaire de classe et une liste de classes parentes. La métaclasse a pour rôle de réunir ces trois paramètres pour construire la classe. La plupart des langages orientés objet fournissent une implémentation par défaut. La particularité de Python est la possibilité de créer des métaclasses personnalisées. La plupart des utilisateurs n'auront jamais besoin de cet outil, mais lorsque le besoin survient, les métaclasses offrent des solutions élégantes et puissantes. Elles sont utilisées pour journaliser les accès à des propriétés, rendre sûrs les environnements *multi-threads*, suivre la création d'objets, implémenter des singletons et bien d'autres tâches.

Plus d'informations sont disponibles dans : *metaclasses*.

méthode Fonction définie à l'intérieur d'une classe. Lorsqu'elle est appelée comme un attribut d'une instance de cette classe, la méthode reçoit l'instance en premier *argument* (qui, par convention, est habituellement nommé `self`). Voir *function* et *nested scope*.

ordre de résolution des méthodes L'ordre de résolution des méthodes (*MRO* pour *Method Resolution Order* en anglais) est, lors de la recherche d'un attribut dans les classes parentes, la façon dont l'interpréteur Python classe ces classes parentes. Voir [The Python 2.3 Method Resolution Order](#) pour plus de détails sur l'algorithme utilisé par l'interpréteur Python depuis la version 2.3.

module Objet utilisé pour organiser une portion unitaire de code en Python. Les modules ont un espace de nommage et peuvent contenir n'importe quels objets Python. Charger des modules est appelé *importer*.

Voir aussi *paquet*.

spécificateur de module Espace de nommage contenant les informations, relatives à l'importation, utilisées pour charger un module. C'est une instance de la classe `importlib.machinery.ModuleSpec`.

MRO Voir *ordre de résolution des méthodes*.

muable Un objet muable peut changer de valeur tout en gardant le même `id()`. Voir aussi *immuable*.

n-uplet nommé (*named-tuple* en anglais) Classe qui, comme un *n-uplet* (*tuple* en anglais), a ses éléments accessibles par leur indice. Et en plus, les éléments sont accessibles par leur nom. Par exemple, `time.localtime()` donne un objet ressemblant à un *n-uplet*, dont `year` est accessible par son indice : `t[0]` ou par son nom : `t.tm_year`. Un *n-uplet nommé* peut être un type natif tel que `time.struct_time` ou il peut être construit comme une simple classe. Un *n-uplet nommé* complet peut aussi être créé via la fonction `collections.namedtuple()`. Cette dernière approche fournit automatiquement des fonctionnalités supplémentaires, tel qu'une représentation lisible comme `Employee(name='jones', title='programmer')`.

espace de nommage L'endroit où une variable est stockée. Les espaces de nommage sont implémentés avec des dictionnaires. Il existe des espaces de nommage globaux, natifs ou imbriqués dans les objets (dans les méthodes). Les espaces de nommage favorisent la modularité car ils permettent d'éviter les conflits de noms. Par exemple, les fonctions `builtins.open` et `os.open()` sont différenciées par leurs espaces de nom. Les espaces de nommage aident aussi à la lisibilité et la maintenabilité en rendant clair quel module implémente une fonction. Par exemple, écrire `random.seed()` ou `itertools.islice()` affiche clairement que ces fonctions sont implémentées respectivement dans les modules `random` et `itertools`.

paquet-espace de nommage Un *paquet* tel que défini dans la [PEP 421](#) qui ne sert qu'à contenir des sous-paquets. Les paquets-espace de nommage peuvent n'avoir aucune représentation physique et, plus spécifiquement, ne sont pas comme un *paquet classique* puisqu'ils n'ont pas de fichier `__init__.py`.

Voir aussi *module*.

portée imbriquée Possibilité de faire référence à une variable déclarée dans une définition englobante. Typiquement, une fonction définie à l'intérieur d'une autre fonction a accès aux variables de cette dernière. Souvenez-vous cependant que cela ne fonctionne que pour accéder à des variables, pas pour les assigner. Les variables locales

sont lues et assignées dans l'espace de nommage le plus proche. Tout comme les variables globales qui sont stockés dans l'espace de nommage global, le mot clef `nonlocal` permet d'écrire dans l'espace de nommage dans lequel est déclarée la variable.

nouvelle classe Ancien nom pour l'implémentation actuelle des classes, pour tous les objets. Dans les anciennes versions de Python, seules les nouvelles classes pouvaient utiliser les nouvelles fonctionnalités telles que `__slots__`, les descripteurs, les propriétés, `__getattr__()`, les méthodes de classe et les méthodes statiques.

objet N'importe quelle donnée comportant des états (sous forme d'attributs ou d'une valeur) et un comportement (des méthodes). C'est aussi (`object`) l'ancêtre commun à absolument toutes les *nouvelles classes*.

paquet *module* Python qui peut contenir des sous-modules ou des sous-paquets. Techniquement, un paquet est un module qui possède un attribut `__path__`.

Voir aussi *paquet classique* et *namespace package*.

paramètre Entité nommée dans la définition d'une *fonction* (ou méthode), décrivant un *argument* (ou dans certains cas des arguments) que la fonction accepte. Il existe cinq sortes de paramètres :

- *positional-or-keyword* : l'argument peut être passé soit par sa *position*, soit en tant que *argument nommé*. C'est le type de paramètre par défaut. Par exemple, *foo* et *bar* dans l'exemple suivant :

```
def func(foo, bar=None): ...
```

- *positional-only* : l'argument ne peut être donné que par sa position. Python n'a pas de syntaxe pour déclarer de tels paramètres, cependant des fonctions natives, comme *abs()*, en utilisent.
- *keyword-only* : l'argument ne peut être fourni que nommé. Les paramètres *keyword-only* peuvent être définis en utilisant un seul paramètre *var-positional*, ou en ajoutant une étoile (*) seule dans la liste des paramètres avant eux. Par exemple, *kw_only1* et *kw_only2* dans le code suivant :

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional* : une séquence d'arguments positionnels peut être fournie (en plus de tous les arguments positionnels déjà acceptés par d'autres paramètres). Un tel paramètre peut être défini en préfixant son nom par une *. Par exemple *args* ci-après :

```
def func(*args, **kwargs): ...
```

- *var-keyword* : une quantité arbitraire d'arguments peut être passée, chacun étant nommé (en plus de tous les arguments nommés déjà acceptés par d'autres paramètres). Un tel paramètre est défini en préfixant le nom du paramètre par **. Par exemple, *kwargs* ci-dessus.

Les paramètres peuvent spécifier des arguments obligatoires ou optionnels, ainsi que des valeurs par défaut pour les arguments optionnels.

Voir aussi *argument* dans le glossaire, la question sur la différence entre les arguments et les paramètres dans la FAQ, la classe `inspect.Parameter`, la section *function* et la **PEP 362**.

entrée de chemin Emplacement dans le *chemin des importations* (*import path* en anglais, d'où le *path*) que le *chercheur basé sur les chemins* consulte pour trouver des modules à importer.

chercheur de chemins *chercheur* renvoyé par un appelable sur un `sys.path_hooks` (c'est-à-dire un *point d'entrée pour la recherche dans path*) qui sait où trouver des modules lorsqu'on lui donne une *entrée de path*.

Voir `importlib.abc.PathEntryFinder` pour les méthodes qu'un chercheur d'entrée dans *path* doit implémenter.

point d'entrée pour la recherche dans path Appelable dans la liste `sys.path_hook` qui donne un *chercheur d'entrée dans path* s'il sait où trouver des modules pour une *entrée dans path* donnée.

chercheur basé sur les chemins L'un des *chercheurs dans les méta-chemins* par défaut qui cherche des modules dans un *chemin des importations*.

objet simili-chemin Objet représentant un chemin du système de fichiers. Un objet simili-chemin est un objet *str* ou un objet *bytes* représentant un chemin ou un objet implémentant le protocole `os.PathLike`. Un objet qui accepte le protocole `os.PathLike` peut être converti en un chemin *str* ou *bytes* du système de fichiers en appelant la fonction `os.fspath()`. `os.fsdecode()` et `os.fsencode()` peuvent être utilisées, respectivement, pour garantir un résultat de type *str* ou *bytes* à la place. A été Introduit par la **PEP 519**.

PEP *Python Enhancement Proposal* (Proposition d'amélioration Python). Un PEP est un document de conception fournissant des informations à la communauté Python ou décrivant une nouvelle fonctionnalité pour Python, ses processus ou son environnement. Les PEP doivent fournir une spécification technique concise et une justification des fonctionnalités proposées.

Les PEPs sont censés être les principaux mécanismes pour proposer de nouvelles fonctionnalités majeures, pour recueillir les commentaires de la communauté sur une question et pour documenter les décisions de conception qui sont intégrées en Python. L'auteur du PEP est responsable de l'établissement d'un consensus au sein de la communauté et de documenter les opinions contradictoires.

Voir [PEP 1](#).

portion Jeu de fichiers dans un seul dossier (pouvant être stocké sous forme de fichier zip) qui contribue à l'espace de nommage d'un paquet, tel que défini dans la [PEP 420](#).

argument positionnel Voir *argument*.

API provisoire Une API provisoire est une API qui n'offre aucune garantie de rétrocompatibilité (la bibliothèque standard exige la rétrocompatibilité). Bien que des changements majeurs d'une telle interface ne soient pas attendus, tant qu'elle est étiquetée provisoire, des changements cassant la rétrocompatibilité (y compris sa suppression complète) peuvent survenir si les développeurs principaux le jugent nécessaire. Ces modifications ne surviendront que si de sérieux problèmes sont découverts et qu'ils n'avaient pas été identifiés avant l'ajout de l'API.

Même pour les API provisoires, les changements cassant la rétrocompatibilité sont considérés comme des « solutions de dernier recours ». Tout ce qui est possible sera fait pour tenter de résoudre les problèmes en conservant la rétrocompatibilité.

Ce processus permet à la bibliothèque standard de continuer à évoluer avec le temps, sans se bloquer longtemps sur des erreurs d'architecture. Voir la [PEP 411](#) pour plus de détails.

paquet provisoire Voir *provisional API*.

Python 3000 Surnom donné à la série des Python 3.x (très vieux surnom donné à l'époque où Python 3 représentait un futur lointain). Aussi abrégé *Py3k*.

Pythonique Idée, ou bout de code, qui colle aux idiomes de Python plutôt qu'aux concepts communs rencontrés dans d'autres langages. Par exemple, il est idiomatique en Python de parcourir les éléments d'un itérable en utilisant `for`. Beaucoup d'autres langages n'ont pas cette possibilité, donc les gens qui ne sont pas habitués à Python utilisent parfois un compteur numérique à la place :

```
for i in range(len(food)) :
    print(food[i])
```

Plutôt qu'utiliser la méthode, plus propre et élégante, donc *Pythonique* :

```
for piece in food:
    print(piece)
```

nom qualifié Nom, comprenant des points, montrant le « chemin » de l'espace de nommage global d'un module vers une classe, fonction ou méthode définie dans ce module, tel que défini dans la [PEP 3155](#). Pour les fonctions et classes de premier niveau, le nom qualifié est le même que le nom de l'objet :

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```


Lorsqu'il est utilisé pour nommer des modules, le *nom qualifié complet* (*fully qualified name* - *FQN* en anglais) signifie le chemin complet (séparé par des points) vers le module, incluant tous les paquets parents. Par exemple : `email.mime.text` :

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

nombre de références Nombre de références à un objet. Lorsque le nombre de références à un objet descend à zéro, l'objet est désalloué. Le comptage de référence n'est généralement pas visible dans le code Python, mais c'est un élément clé de l'implémentation *CPython*. Le module *sys* définit une fonction *getrefcount()* que les développeurs peuvent utiliser pour obtenir le nombre de références à un objet donné.

paquet classique *paquet* traditionnel, tel qu'un dossier contenant un fichier `__init__.py`.

Voir aussi *paquet-espace de nommage*.

__slots__ Déclaration dans une classe qui économise de la mémoire en pré-allouant de l'espace pour les attributs des instances et qui élimine le dictionnaire (des attributs) des instances. Bien que populaire, cette technique est difficile à maîtriser et devrait être réservée à de rares cas où un grand nombre d'instances dans une application devient un sujet critique pour la mémoire.

séquence *itérable* qui offre un accès efficace à ses éléments par un indice sous forme de nombre entier via la méthode spéciale `__getitem__()` et qui définit une méthode `__len__()` donnant sa taille. Voici quelques séquences natives : *list*, *str*, *tuple*, et *bytes*. Notez que *dict* possède aussi une méthode `__getitem__()` et une méthode `__len__()`, mais il est considéré comme un *mapping* plutôt qu'une séquence, car ses accès se font par une clé arbitraire *immutable* plutôt qu'un nombre entier.

La classe abstraite de base *collections.abc.Sequence* définit une interface plus riche qui va au-delà des simples `__getitem__()` et `__len__()`, en ajoutant `count()`, `index()`, `__contains__()` et `__reversed__()`. Les types qui implémentent cette interface étendue peuvent s'enregistrer explicitement en utilisant `register()`.

distribution simple Forme de distribution, comme les *fonction génériques*, où l'implémentation est choisie en fonction du type d'un seul argument.

tranche (*slice* en anglais), un objet contenant habituellement une portion de *séquence*. Une tranche est créée en utilisant la notation `[]` avec des `:` entre les nombres lorsque plusieurs sont fournis, comme dans `variable_name[1:3:5]`. Cette notation utilise des objets *slice* en interne.

méthode spéciale (*special method* en anglais) Méthode appelée implicitement par Python pour exécuter une opération sur un type, comme une addition. De telles méthodes ont des noms commençant et terminant par des doubles tirets bas. Les méthodes spéciales sont documentées dans *specialnames*.

instruction Une instruction (*statement* en anglais) est un composant d'un « bloc » de code. Une instruction est soit une *expression*, soit une ou plusieurs constructions basées sur un mot-clé, comme `if`, `while` ou `for`.

struct sequence A tuple with named elements. Struct sequences expose an interface similar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `__make()` or `__asdict()`. Examples of struct sequences include *sys.float_info* and the return value of *os.stat()*.

encodage de texte Codec (codeur-décodeur) qui convertit des chaînes de caractères Unicode en octets (classe *bytes*).

fichier texte *file object* capable de lire et d'écrire des objets *str*. Souvent, un fichier texte (*text file* en anglais) accède en fait à un flux de donnée en octets et gère l'*text encoding* automatiquement. Des exemples de fichiers textes sont les fichiers ouverts en mode texte ('r' ou 'w'), *sys.stdin*, *sys.stdout* et les instances de *io.StringIO*.

Voir aussi *binary file* pour un objet fichier capable de lire et d'écrire *bytes-like objects*.

chaîne entre triple guillemets Chaîne qui est délimitée par trois guillemets simples (') ou trois guillemets doubles ("). Bien qu'elle ne fournisse aucune fonctionnalité qui ne soit pas disponible avec une chaîne entre guillemets, elle est utile pour de nombreuses raisons. Elle vous autorise à insérer des guillemets simples et doubles dans une chaîne sans avoir à les protéger et elle peut s'étendre sur plusieurs lignes sans avoir à terminer chaque ligne par un \. Elle est ainsi particulièrement utile pour les chaînes de documentation (*docstrings*).

type Le type d'un objet Python détermine quel genre d'objet c'est. Tous les objets ont un type. Le type d'un objet peut être obtenu via son attribut `__class__` ou via `type(obj)`.

alias de type Synonyme d'un type, créé en affectant le type à un identifiant.

Les alias de types sont utiles pour simplifier les *indications de types*. Par exemple :

```
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

pourrait être rendu plus lisible comme ceci :

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

Voir *typing* et **PEP 484**, qui décrivent cette fonctionnalité.

indication de type Le *annotation* qui spécifie le type attendu pour une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour.

Les indications de type sont facultatives et ne sont pas indispensables à l'interpréteur Python, mais elles sont utiles aux outils d'analyse de type statique et aident les IDE à compléter et à réusiner (*code refactoring* en anglais) le code.

Les indicateurs de type de variables globales, d'attributs de classe et de fonctions, mais pas de variables locales, peuvent être consultés en utilisant `typing.get_type_hints()`.

Voir *typing* et **PEP 484**, qui décrivent cette fonctionnalité.

retours à la ligne universels Une manière d'interpréter des flux de texte dans lesquels sont reconnues toutes les fins de ligne suivantes : la convention Unix `'\n'`, la convention Windows `'\r\n'` et l'ancienne convention Macintosh `'\r'`. Voir la **PEP 278** et la **PEP 3116**, ainsi que la fonction `bytes.splitlines()` pour d'autres usages.

annotation de variable *annotation* d'une variable ou d'un attribut de classe.

Lorsque vous annotez une variable ou un attribut de classe, l'affectation est facultative :

```
class C:
    field: 'annotation'
```

Les annotations de variables sont généralement utilisées pour des *indications de types* : par exemple, cette variable devrait prendre des valeurs de type `int` :

```
count: int = 0
```

La syntaxe d'annotation de la variable est expliquée dans la section *annassign*.

Reportez-vous à *function annotation*, à la **PEP 484** et à la **PEP 526** qui décrivent cette fonctionnalité.

environnement virtuel Environnement d'exécution isolé (en mode coopératif) qui permet aux utilisateurs de Python et aux applications d'installer et de mettre à jour des paquets sans interférer avec d'autres applications Python fonctionnant sur le même système.

Voir aussi *venv*.

machine virtuelle Ordinateur défini entièrement par du logiciel. La machine virtuelle (*virtual machine*) de Python exécute le *bytecode* produit par le compilateur de *bytecode*.

Le zen de Python Liste de principes et de préceptes utiles pour comprendre et utiliser le langage. Cette liste peut être obtenue en tapant `« import this »` dans une invite Python interactive.

À propos de ces documents

Ces documents sont générés à partir de sources en [reStructuredText](#) par [Sphinx](#), un analyseur de documents spécialement conçu pour la documentation Python.

Le développement de la documentation et de ses outils est entièrement basé sur le volontariat, tout comme Python. Si vous voulez contribuer, allez voir la page [reporting-bugs](#) qui contient des informations pour vous y aider. Les nouveaux volontaires sont toujours les bienvenus !

Merci beaucoup à :

- Fred L. Drake, Jr., créateur des outils originaux de la documentation Python et rédacteur de la plupart de son contenu ;
- le projet [Docutils](#) pour avoir créé *reStructuredText* et la suite d'outils *Docutils* ;
- Fredrik Lundh pour son projet [Alternative Python Reference](#), dont Sphinx a pris beaucoup de bonnes idées.

B.1 Contributeurs de la documentation Python

De nombreuses personnes ont contribué au langage Python, à sa bibliothèque standard et à sa documentation. Consultez [Misc/ACKS](#) dans les sources de la distribution Python pour avoir une liste partielle des contributeurs.

Ce n'est que grâce aux suggestions et contributions de la communauté Python que Python a une documentation si merveilleuse — Merci !

Histoire et licence

C.1 Histoire du logiciel

Python a été créé au début des années 1990 par Guido van Rossum, au Stichting Mathematisch Centrum (CWI, voir <https://www.cwi.nl/>) au Pays-Bas en tant que successeur d'un langage appelé ABC. Guido est l'auteur principal de Python, bien qu'il inclut de nombreuses contributions de la part d'autres personnes.

En 1995, Guido continua son travail sur Python au Corporation for National Research Initiatives (CNRI, voir <https://www.cnri.reston.va.us/>) de Reston, en Virginie, d'où il diffusa plusieurs versions du logiciel.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <https://www.zope.org/>). In 2001, the Python Software Foundation (PSF, see <https://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

Toutes les versions de Python sont Open Source (voir <https://www.opensource.org/> pour la définition d'Open Source). Historiquement, la plupart, mais pas toutes, des versions de Python ont également été compatible avec la GPL, le tableau ci-dessous résume les différentes versions.

Version	Dérivé de	Année	Propriétaire	Compatible avec la GPL ?
0.9.0 à 1.2	n/a	1991-1995	CWI	oui
1.3 à 1.5.2	1.2	1995-1999	CNRI	oui
1.6	1.5.2	2000	CNRI	non
2.0	1.6	2000	BeOpen.com	non
1.6.1	1.6	2001	CNRI	non
2.1	2.0+1.6.1	2001	PSF	non
2.0.1	2.0+1.6.1	2001	PSF	oui
2.1.1	2.1+2.0.1	2001	PSF	oui
2.1.2	2.1.1	2002	PSF	oui
2.1.3	2.1.2	2002	PSF	oui
2.2 et ultérieure	2.1.1	2001-maintenant	PSF	oui

Note : Compatible GPL ne signifie pas que nous distribuons Python sous licence GPL. Toutes les licences Python, excepté la licence GPL, vous permettent la distribution d'une version modifiée sans rendre open source ces changements. La licence « compatible GPL » rend possible la diffusion de Python avec un autre logiciel qui est lui, diffusé sous la licence GPL ; les licences « non-compatibles GPL » ne le peuvent pas.

Merci aux nombreux bénévoles qui ont travaillé sous la direction de Guido pour rendre ces versions possibles.

C.2 Conditions générales pour accéder à, ou utiliser, Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.6.11

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),
→and
the Individual or Organization ("Licensee") accessing and otherwise using
→Python
3.6.11 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby
grants Licensee a nonexclusive, royalty-free, world-wide license to
→reproduce,
analyze, test, perform and/or display publicly, prepare derivative works,
distribute, and otherwise use Python 3.6.11 alone or in any derivative
version, provided, however, that PSF's License Agreement and PSF's notice
→of
copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All
→Rights
Reserved" are retained in Python 3.6.11 alone or in any derivative version
prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or
incorporates Python 3.6.11 or any part thereof, and wants to make the
derivative work available to others as provided herein, then Licensee
→hereby
agrees to include in any such work a brief summary of the changes made to
→Python
3.6.11.
4. PSF is making Python 3.6.11 available to Licensee on an "AS IS" basis.
PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF
EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION
→OR
WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT
→THE
USE OF PYTHON 3.6.11 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.6.11
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT
→OF
MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.6.11, OR ANY
→DERIVATIVE

THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 3.6.11, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0

LICENCE D'UTILISATION LIBRE BEOPEN PYTHON VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any

(suite sur la page suivante)

(suite de la page précédente)

third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.3 LICENCE D'UTILISATION CNRI POUR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python

(suite sur la page suivante)

(suite de la page précédente)

1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3 Licences et remerciements pour les logiciels tiers

Cette section est une liste incomplète mais grandissante de licences et remerciements pour les logiciels tiers incorporés dans la distribution de Python.

C.3.1 Mersenne twister

Le module `_random` inclut du code construit à partir d'un téléchargement depuis <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. Voici mot pour mot les commentaires du code original :

A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)`
or `init_by_array(init_key, key_length)`.

(suite sur la page suivante)

(suite de la page précédente)

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote
products derived from this software without specific prior written
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

C.3.2 Interfaces de connexion (*sockets*)

Le module *socket* utilise les fonctions `getaddrinfo()` et `getnameinfo()` codées dans des fichiers source séparés et provenant du projet WIDE : <http://www.wide.ad.jp/>.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
may be used to endorse or promote products derived from this software
without specific prior written permission.

(suite sur la page suivante)

(suite de la page précédente)

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Virgule flottante et contrôle d'exception

Le code source pour le module *fpectl* inclut la note suivante :

```
-----
/                               Copyright (c) 1996.                               \
|                               The Regents of the University of California.          |
|                               All rights reserved.                                |
|                                                                                 |
|  Permission to use, copy, modify, and distribute this software for               |
|  any purpose without fee is hereby granted, provided that this en-               |
|  tire notice is included in all copies of any software which is or               |
|  includes a copy or modification of this software and in all                     |
|  copies of the supporting documentation for such software.                       |
|                                                                                 |
|  This work was produced at the University of California, Lawrence                 |
|  Livermore National Laboratory under contract no. W-7405-ENG-48                  |
|  between the U.S. Department of Energy and The Regents of the                   |
|  University of California for the operation of UC LLNL.                         |
|                                                                                 |
|                               DISCLAIMER                                           |
|                                                                                 |
|  This software was prepared as an account of work sponsored by an                |
|  agency of the United States Government. Neither the United States               |
|  Government nor the University of California nor any of their em-               |
|  ployees, makes any warranty, express or implied, or assumes any                 |
|  liability or responsibility for the accuracy, completeness, or                   |
|  usefulness of any information, apparatus, product, or process                   |
|  disclosed, or represents that its use would not infringe                       |
|  privately-owned rights. Reference herein to any specific commer-               |
|  cial products, process, or service by trade name, trademark,                   |
|  manufacturer, or otherwise, does not necessarily constitute or                 |
|  imply its endorsement, recommendation, or favoring by the United               |
|  States Government or the University of California. The views and                |
|  opinions of authors expressed herein do not necessarily state or                |
|  reflect those of the United States Government or the University                 |
|  of California, and shall not be used for advertising or product                 |
|  endorsement purposes.                                                           |
\-----
```

C.3.4 Interfaces de connexion asynchrones

Les modules `asynchat` et `asyncore` contiennent la note suivante :

```
Copyright 1996 by Sam Rushing
```

```
    All Rights Reserved
```

```
Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.
```

```
SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Gestion de témoin (*cookie*)

Le module `http.cookies` contient la note suivante :

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>
```

```
    All Rights Reserved
```

```
Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.
```

```
Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.6 Traçage d'exécution

Le module `trace` contient la note suivante :

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.7 Les fonctions UUencode et UUdecode

Le module `uu` contient la note suivante :

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
    All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
```

(suite sur la page suivante)

(suite de la page précédente)

```
version is still 5 times faster, though.  
- Arguments more compliant with Python standard
```

C.3.8 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*)

Le module `xmlrpc.client` contient la note suivante :

```
The XML-RPC client interface is  
  
Copyright (c) 1999-2002 by Secret Labs AB  
Copyright (c) 1999-2002 by Fredrik Lundh  
  
By obtaining, using, and/or copying this software and/or its  
associated documentation, you agree that you have read, understood,  
and will comply with the following terms and conditions:  
  
Permission to use, copy, modify, and distribute this software and  
its associated documentation for any purpose and without fee is  
hereby granted, provided that the above copyright notice appears in  
all copies, and that both that copyright notice and this permission  
notice appear in supporting documentation, and that the name of  
Secret Labs AB or the author not be used in advertising or publicity  
pertaining to distribution of the software without specific, written  
prior permission.  
  
SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD  
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-  
ABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR  
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY  
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,  
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS  
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE  
OF THIS SOFTWARE.
```

C.3.9 test_epoll

Le module `test_epoll` contient la note suivante :

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.  
  
Permission is hereby granted, free of charge, to any person obtaining  
a copy of this software and associated documentation files (the  
"Software"), to deal in the Software without restriction, including  
without limitation the rights to use, copy, modify, merge, publish,  
distribute, sublicense, and/or sell copies of the Software, and to  
permit persons to whom the Software is furnished to do so, subject to  
the following conditions:  
  
The above copyright notice and this permission notice shall be  
included in all copies or substantial portions of the Software.  
  
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,  
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(suite sur la page suivante)

(suite de la page précédente)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.10 Select queue

Le module `select` contient la note suivante pour l'interface `kqueue` :

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.11 SipHash24

Le fichier `Python/pyhash.c` contient une implémentation par Marek Majkowski de l'algorithme *SipHash24* de Dan Bernstein. Il contient la note suivante :

```
<MIT License>
Copyright (c) 2013 Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>
```

(suite sur la page suivante)

(suite de la page précédente)

```
Original location:
    https://github.com/majek/csiphash/

Solution inspired by code from:
    Samuel Neves (supercop/crypto_auth/siphhash24/little)
    djb (supercop/crypto_auth/siphhash24/little2)
    Jean-Philippe Aumasson (https://131002.net/siphhash/siphhash24.c)
```

C.3.12 *strtod* et *dtoa*

Le fichier `Python/dtoa.c`, qui fournit les fonctions `dtoa` et `strtod` pour la conversion de *double*s C vers et depuis les chaînes, est tiré d'un fichier du même nom par David M. Gay, actuellement disponible sur <http://www.netlib.org/fp/>. Le fichier original, tel que récupéré le 16 mars 2009, contient la licence suivante :

```
/* *****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 * *****/
```

C.3.13 OpenSSL

Les modules `hashlib`, `posix`, `ssl`, et `crypt` utilisent la bibliothèque OpenSSL pour améliorer les performances, si elle est disponible via le système d'exploitation. Aussi les outils d'installation sur Windows et Mac OS X peuvent inclure une copie des bibliothèques d'OpenSSL, donc on colle une copie de la licence d'OpenSSL ici :

```
LICENSE ISSUES
=====

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
-----

/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.
```

(suite sur la page suivante)

(suite de la page précédente)

```

*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in
*    the documentation and/or other materials provided with the
*    distribution.
*
* 3. All advertising materials mentioning features or use of this
*    software must display the following acknowledgment:
*    "This product includes software developed by the OpenSSL Project
*    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
*
* 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
*    endorse or promote products derived from this software without
*    prior written permission. For written permission, please contact
*    openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
*    nor may "OpenSSL" appear in their names without prior written
*    permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*    acknowledgment:
*    "This product includes software developed by the OpenSSL Project
*    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.

```

(suite sur la page suivante)

```

*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to. The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code. The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*    must display the following acknowledgement:
*    "This product includes cryptographic software written by
*    Eric Young (eay@cryptsoft.com)"
*    The word 'cryptographic' can be left out if the rouines from the library
*    being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*    the apps directory (application code) you must include an acknowledgement:
*    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/

```

C.3.14 expat

Le module `pyexpat` est compilé avec une copie des sources d'*expat*, sauf si la compilation est configurée avec `--with-system-expat` :

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.15 libffi

Le module `_ctypes` est compilé en utilisant une copie des sources de la *libffi*, sauf si la compilation est configurée avec `--with-system-libffi` :

```
Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.16 zlib

Le module `zlib` est compilé en utilisant une copie du code source de `zlib` si la version de `zlib` trouvée sur le système est trop vieille pour être utilisée :

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly          Mark Adler
jloup@gzip.org            madler@alumni.caltech.edu
```

C.3.17 cfuhash

L'implémentation des dictionnaires, utilisée par le module `tracemalloc` est basée sur le projet `cfuhash` :

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above
  copyright notice, this list of conditions and the following
  disclaimer in the documentation and/or other materials provided
  with the distribution.

* Neither the name of the author nor the names of its
  contributors may be used to endorse or promote products derived
  from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
```

(suite sur la page suivante)

(suite de la page précédente)

```
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.18 libmpdec

Le module `_decimal` est construit en incluant une copie de la bibliothèque *libmpdec*, sauf si elle est compilée avec `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```


ANNEXE D

Copyright

Python et cette documentation sont :

Copyright © 2001-2020 Python Software Foundation. All rights reserved.

Copyright © 2000 *BeOpen.com*. Tous droits réservés.

Copyright © 1995-2000 *Corporation for National Research Initiatives*. Tous droits réservés.

Copyright © 1991-1995 *Stichting Mathematisch Centrum*. Tous droits réservés.

Voir [Histoire et licence](#) pour des informations complètes concernant la licence et les permissions.

Bibliographie

- [Frie09] *Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O'Reilly Media, 2009.* La troisième édition de ce livre ne couvre plus du tout Python, mais la première version explique en détails comment écrire de bonnes expressions rationnelles.
- [C99] *ISO/IEC 9899 :1999.* « Langages de programmation – C. » Un texte public de ce standard est disponible à <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf>.

—
__future__, 1587
__main__, 1558
_dummy_thread, 796
_thread, 794

a

abc, 1575
aifc, 1226
argparse, 581
array, 228
ast, 1643
asynchat, 919
asyncio, 857
asyncore, 915
atexit, 1579
audioop, 1223

b

base64, 1015
bdb, 1483
binascii, 1019
binhex, 1018
bisect, 226
builtins, 1557
bz2, 445

c

calendar, 199
cgi, 1088
cgitb, 1095
chunk, 1234
cmath, 278
cmd, 1293
code, 1611
codecs, 151
codeop, 1613
collections, 202
collections.abc, 218

colorsys, 1235
compileall, 1660
concurrent.futures, 768
configparser, 478
contextlib, 1563
copy, 242
copyreg, 410
cProfile, 1498
crypt (*Unix*), 1705
csv, 471
ctypes, 681
curses (*Unix*), 650
curses.ascii, 669
curses.panel, 671
curses.textpad, 667

d

datetime, 169
dbm, 415
dbm.dumb, 418
dbm.gnu (*Unix*), 416
dbm.ndbm (*Unix*), 417
decimal, 281
difflib, 122
dis, 1662
distutils, 1521
doctest, 1363
dummy_threading, 793

e

email, 931
email.charset, 979
email.contentmanager, 958
email.encoders, 981
email.errors, 952
email.generator, 943
email.header, 977
email.headerregistry, 953
email.iterators, 984
email.message, 932

email.mime, 974
email.parser, 940
email.policy, 946
email.utils, 982
encodings.idna, 166
encodings.mbc, 166
encodings.utf_8_sig, 167
ensurepip, 1522
enum, 251
errno, 675

f

faulthandler, 1487
fcntl (*Unix*), 1709
filecmp, 379
fileinput, 372
fnmatch, 386
formatter, 1677
fpectl (*Unix*), 1608
fractions, 306
ftplib, 1138
functools, 336

g

gc, 1588
getopt, 612
getpass, 650
gettext, 1243
glob, 385
grp (*Unix*), 1704
gzip, 443

h

hashlib, 503
heapq, 222
hmac, 514
html, 1023
html.entities, 1028
html.parser, 1024
http, 1130
http.client, 1132
http.cookiejar, 1190
http.cookies, 1186
http.server, 1181

i

imaplib, 1146
imghdr, 1236
imp, 1745
importlib, 1623
importlib.abc, 1625
importlib.machinery, 1630
importlib.util, 1634
inspect, 1591

io, 562
ipaddress, 1211
itertools, 321

j

json, 985
json.tool, 993

k

keyword, 1653

l

lib2to3, 1473
linecache, 387
locale, 1251
logging, 614
logging.config, 629
logging.handlers, 638
lzma, 448

m

macpath, 395
mailbox, 995
mailcap, 994
marshal, 413
math, 272
mimetypes, 1012
mmap, 926
modulefinder, 1619
msilib (*Windows*), 1683
msvcrt (*Windows*), 1689
multiprocessing, 727
multiprocessing.connection, 755
multiprocessing.dummy, 759
multiprocessing.managers, 746
multiprocessing.pool, 752
multiprocessing.sharedctypes, 744

n

netrc, 495
nis (*Unix*), 1716
nntplib, 1152
numbers, 269

o

operator, 342
optparse, 1719
os, 519
os.path, 367
ossaudiodev (*Linux, FreeBSD*), 1237

p

parser, 1639

pathlib, 351
 pdb, 1489
 pickle, 397
 pickletools, 1675
 pipes (*Unix*), 1711
 pkgutil, 1617
 platform, 672
 plistlib, 499
 poplib, 1143
 posix (*Unix*), 1701
 pprint, 243
 profile, 1498
 pstats, 1499
 pty (*Linux*), 1708
 pwd (*Unix*), 1702
 py_compile, 1659
 pycldr, 1657
 pydoc, 1362

q

queue, 791
 quopri, 1021

r

random, 308
 re, 102
 readline (*Unix*), 139
 reprlib, 249
 resource (*Unix*), 1712
 rlcompleter, 144
 runpy, 1621

s

sched, 789
 secrets, 515
 select, 847
 selectors, 854
 shelve, 411
 shlex, 1298
 shutil, 388
 signal, 921
 site, 1605
 smtpd, 1165
 smtplib, 1159
 sndhdr, 1237
 socket, 797
 socketserver, 1173
 spwd (*Unix*), 1703
 sqlite3, 419
 ssl, 817
 stat, 374
 statistics, 315
 string, 91
 stringprep, 138

struct, 145
 subprocess, 774
 sunau, 1229
 symbol, 1651
 symtable, 1649
 sys, 1539
 sysconfig, 1554
 syslog (*Unix*), 1717

t

tabnanny, 1657
 tarfile, 461
 telnetlib, 1168
 tempfile, 381
 termios (*Unix*), 1706
 test, 1473
 test.support, 1476
 textwrap, 132
 threading, 715
 time, 573
 timeit, 1503
 tkinter, 1303
 tkinter.scrolledtext (*Tk*), 1335
 tkinter.tix, 1330
 tkinter.ttk, 1314
 token, 1651
 tokenize, 1653
 trace, 1508
 traceback, 1581
 tracemalloc, 1510
 tty (*Unix*), 1708
 turtle, 1261
 turtledemo, 1291
 types, 239
 typing, 1347

u

unicodedata, 136
 unittest, 1386
 unittest.mock, 1413
 urllib, 1104
 urllib.error, 1128
 urllib.parse, 1122
 urllib.request, 1104
 urllib.response, 1121
 urllib.robotparser, 1129
 uu, 1021
 uuid, 1170

v

venv, 1523

w

warnings, 1558

- wave, 1231
- weakref, 231
- webbrowser, 1085
- winreg (*Windows*), 1690
- winsound (*Windows*), 1698
- wsgiref, 1095
- wsgiref.handlers, 1101
- wsgiref.headers, 1098
- wsgiref.simple_server, 1099
- wsgiref.util, 1096
- wsgiref.validate, 1100

X

- xdrlib, 496
- xml, 1029
- xml.dom, 1045
- xml.dom.minidom, 1056
- xml.dom.pulldom, 1060
- xml.etree.ElementTree, 1030
- xml.parsers.expat, 1074
- xml.parsers.expat.errors, 1081
- xml.parsers.expat.model, 1080
- xml.sax, 1062
- xml.sax.handler, 1064
- xml.sax.saxutils, 1069
- xml.sax.xmlreader, 1070
- xmlrpc.client, 1198
- xmlrpc.server, 1205

Z

- zipapp, 1531
- zipfile, 453
- zipimport, 1615
- zlib, 439

Non alphabétique

- ??
 - in regular expressions, 103
- ..
 - in pathnames, 560
- ..., 1753
 - ellipsis literal, 25, 78
 - in doctests, 1371
 - interpreter prompt, 1368, 1549
 - placeholder, 135, 243, 249
- . (*dot*)
 - in glob-style wildcards, 385
 - in pathnames, 560, 561
 - in printf-style formatting, 49, 62
 - in regular expressions, 103
 - in string formatting, 93
 - in Tkinter, 1306
- ! (*exclamation*)
 - in a command interpreter, 1293
 - in curses module, 670
 - in glob-style wildcards, 385, 386
 - in string formatting, 93
 - in struct format strings, 147
- (*minus*)
 - binary operator, 29
 - in doctests, 1372
 - in glob-style wildcards, 385, 386
 - in printf-style formatting, 50, 63
 - in regular expressions, 104
 - in string formatting, 95
 - unary operator, 29
- ! (*pdb command*), 1495
- ? (*question mark*)
 - in a command interpreter, 1293
 - in argparse module, 594
 - in AST grammar, 1643
 - in glob-style wildcards, 385, 386
 - in regular expressions, 103
 - in SQL statements, 427
 - in struct format strings, 148, 149
 - replacement character, 154
- # (*hash*)
 - comment, 1606
 - in doctests, 1372
 - in printf-style formatting, 50, 63
 - in regular expressions, 109
 - in string formatting, 96
- \$ (*dollar*)
 - environment variables expansion, 369
 - in regular expressions, 103
 - in template strings, 100
 - interpolation in configuration files, 482
- % (*percent*)
 - datetime format, 195, 576, 578
 - environment variables expansion (*Windows*), 369, 1692
 - interpolation in configuration files, 482
 - opérateur, 29
 - printf-style formatting, 49, 62
- & (*ampersand*)
 - opérateur, 30
- (?
 - in regular expressions, 104
- (?!
 - in regular expressions, 105
- (?#
 - in regular expressions, 105
- () (*parentheses*)
 - in printf-style formatting, 49, 62
 - in regular expressions, 104
- (?:
 - in regular expressions, 105
- (<?!
 - in regular expressions, 106
- (<=
 - in regular expressions, 105
- (<P<
 - in regular expressions, 105

in regular expressions, 105
(?P=
in regular expressions, 105
*?
in regular expressions, 103
* (*asterisk*)
in argparse module, 594
in AST grammar, 1643
in glob-style wildcards, 385, 386
in printf-style formatting, 49, 62
in regular expressions, 103
opérateur, 29
**
in glob-style wildcards, 385
opérateur, 29
+?
in regular expressions, 103
+ (*plus*)
binary operator, 29
in argparse module, 594
in doctests, 1372
in printf-style formatting, 50, 63
in regular expressions, 103
in string formatting, 95
unary operator, 29
, (*comma*)
in string formatting, 96
/ (*slash*)
in pathnames, 560, 561
opérateur, 29
//
opérateur, 29
2to3, 1753
: (*colon*)
in SQL statements, 427
in string formatting, 93
path separator (*POSIX*), 561
; (*semicolon*), 561
< (*less*)
in string formatting, 95
in struct format strings, 147
opérateur, 28
<<
opérateur, 30
<=
opérateur, 28
<BLANKLINE>, 1371
!=
opérateur, 28
= (*equals*)
in string formatting, 95
in struct format strings, 147
==
opérateur, 28
> (*greater*)
in string formatting, 95
in struct format strings, 147
opérateur, 28
>=
opérateur, 28
>>
opérateur, 30
>>>, 1753
interpreter prompt, 1368, 1549
@ (*at*)
in struct format strings, 147
[] (*square brackets*)
in glob-style wildcards, 385, 386
in regular expressions, 104
in string formatting, 93
\ (*backslash*)
escape sequence, 154
in pathnames (*Windows*), 560
in regular expressions, 104, 106
\\
in regular expressions, 107
\\A
in regular expressions, 106
\\a
in regular expressions, 107
\\B
in regular expressions, 106
\\b
in regular expressions, 106, 107
\\D
in regular expressions, 107
\\d
in regular expressions, 107
\\f
in regular expressions, 107
\\g
in regular expressions, 111
\\N
escape sequence, 154
in regular expressions, 107
\\n
in regular expressions, 107
\\r
in regular expressions, 107
\\S
in regular expressions, 107
\\s
in regular expressions, 107
\\t
in regular expressions, 107
\\U
escape sequence, 154
in regular expressions, 107

u
 escape sequence, 154
 in regular expressions, 107

v
 in regular expressions, 107

W
 in regular expressions, 107

w
 in regular expressions, 107

x
 escape sequence, 154
 in regular expressions, 107

Z
 in regular expressions, 107

^ (caret)
 in curses module, 670
 in regular expressions, 103, 104
 in string formatting, 95
 marker, 1370, 1581
 opérateur, 30

_ (underscore)
 gettext, 1244
 in string formatting, 96

abs__ () (dans le module operator), 343

add__ () (dans le module operator), 343

and__ () (dans le module operator), 343

bases__ (attribut class), 79

bytes__ () (méthode email.message.EmailMessage), 933

bytes__ () (méthode email.message.Message), 968

call__ () (méthode email.headerregistry.HeaderRegistry), 957

call__ () (méthode weakref.finalize), 234

callback__ (attribut weakref.ref), 233

cause__ (attribut traceback.TracebackException), 1583

ceil__ () (méthode fractions.Fraction), 308

class__ (attribut instance), 79

class__ (attribut unittest.mock.Mock), 1423

code__ (function object attribute), 78

concat__ () (dans le module operator), 344

contains__ () (dans le module operator), 344

contains__ () (méthode email.message.EmailMessage), 934

contains__ () (méthode email.message.Message), 970

contains__ () (méthode mailbox.Mailbox), 998

context__ (attribut traceback.TracebackException), 1583

copy__ () (copy protocol), 243

debug__ (variable de base), 25

deepcopy__ () (copy protocol), 243

del__ () (méthode io.IOBBase), 566

delitem__ () (dans le module operator), 344

delitem__ () (méthode email.message.EmailMessage), 934

delitem__ () (méthode email.message.Message), 970

delitem__ () (méthode mailbox.Mailbox), 996

delitem__ () (méthode mailbox.MH), 1001

dict__ (attribut object), 79

dir__ () (méthode unittest.mock.Mock), 1419

displayhook__ (dans le module sys), 1541

doc__ (attribut types.ModuleType), 240

enter__ () (méthode contextmanager), 76

enter__ () (méthode winreg.PyHKEY), 1698

eq__ () (dans le module operator), 342

eq__ () (instance method), 28

eq__ () (méthode email.charset.Charset), 980

eq__ () (méthode email.header.Header), 978

eq__ () (méthode memoryview), 65

excepthook__ (dans le module sys), 1541

exit__ () (méthode contextmanager), 76

exit__ () (méthode winreg.PyHKEY), 1698

floor__ () (méthode fractions.Fraction), 308

floordiv__ () (dans le module operator), 343

format__, 11

format__ () (méthode datetime.date), 176

format__ () (méthode datetime.datetime), 183

format__ () (méthode datetime.time), 187

fspath__ () (méthode os.PathLike), 521

future__, 1757

future__ (module), 1587

ge__ () (dans le module operator), 342

ge__ () (instance method), 28

getitem__ () (dans le module operator), 345

getitem__ () (méthode email.headerregistry.HeaderRegistry), 957

getitem__ () (méthode email.message.EmailMessage), 934

getitem__ () (méthode email.message.Message), 970

getitem__ () (méthode mailbox.Mailbox), 997

getitem__ () (méthode re.match), 115

getnewargs__ () (méthode object), 403

getnewargs_ex__ () (méthode object), 403

getstate__ () (copy protocol), 407

getstate__ () (méthode object), 403

gt__ () (dans le module operator), 342

gt__ () (instance method), 28

iadd__ () (dans le module operator), 348

iand__ () (dans le module operator), 348

iconcat__ () (dans le module operator), 348

ifloordiv__ () (dans le module operator), 348

ilshift__ () (dans le module operator), 348

imatmul__ () (dans le module operator), 348

imod__ () (dans le module operator), 348

import__ () (dans le module importlib), 1623

import__ () (fonction de base), 23

imul__ () (dans le module operator), 348

- `__index__()` (dans le module *operator*), 343
- `__init__()` (méthode *difflib.HtmlDiff*), 123
- `__init__()` (méthode *logging.Handler*), 618
- `__interactivehook__` (dans le module *sys*), 1547
- `__inv__()` (dans le module *operator*), 343
- `__invert__()` (dans le module *operator*), 343
- `__ior__()` (dans le module *operator*), 348
- `__ipow__()` (dans le module *operator*), 348
- `__irshift__()` (dans le module *operator*), 348
- `__isub__()` (dans le module *operator*), 348
- `__iter__()` (méthode *container*), 34
- `__iter__()` (méthode *iterator*), 35
- `__iter__()` (méthode *mailbox.Mailbox*), 997
- `__iter__()` (méthode *unittest.TestSuite*), 1404
- `__itruediv__()` (dans le module *operator*), 348
- `__ixor__()` (dans le module *operator*), 348
- `__le__()` (dans le module *operator*), 342
- `__le__()` (instance method), 28
- `__len__()` (méthode *email.message.EmailMessage*), 934
- `__len__()` (méthode *email.message.Message*), 970
- `__len__()` (méthode *mailbox.Mailbox*), 998
- `__loader__` (attribut *types.ModuleType*), 240
- `__lshift__()` (dans le module *operator*), 343
- `__lt__()` (dans le module *operator*), 342
- `__lt__()` (instance method), 28
- `__main__`
 - module, 1621, 1622
- `__main__` (module), 1558
- `__matmul__()` (dans le module *operator*), 344
- `__missing__()`, 73
- `__missing__()` (méthode *collections.defaultdict*), 210
- `__mod__()` (dans le module *operator*), 344
- `__mro__` (attribut *class*), 79
- `__mul__()` (dans le module *operator*), 344
- `__name__` (attribut *definition*), 79
- `__name__` (attribut *types.ModuleType*), 240
- `__ne__()` (dans le module *operator*), 342
- `__ne__()` (instance method), 28
- `__ne__()` (méthode *email.charset.Charset*), 980
- `__ne__()` (méthode *email.header.Header*), 978
- `__neg__()` (dans le module *operator*), 344
- `__next__()` (méthode *csv.csvreader*), 476
- `__next__()` (méthode *iterator*), 35
- `__not__()` (dans le module *operator*), 343
- `__or__()` (dans le module *operator*), 344
- `__package__` (attribut *types.ModuleType*), 240
- `__pos__()` (dans le module *operator*), 344
- `__pow__()` (dans le module *operator*), 344
- `__qualname__` (attribut *definition*), 79
- `__reduce__()` (méthode *object*), 403
- `__reduce_ex__()` (méthode *object*), 404
- `__repr__()` (méthode *multiprocessing.managers.BaseProxy*), 752
- `__repr__()` (méthode *netrc.netrc*), 495
- `__round__()` (méthode *fractions.Fraction*), 308
- `__rshift__()` (dans le module *operator*), 344
- `__setitem__()` (dans le module *operator*), 345
- `__setitem__()` (méthode *email.message.EmailMessage*), 934
- `__setitem__()` (méthode *email.message.Message*), 970
- `__setitem__()` (méthode *mailbox.Mailbox*), 996
- `__setitem__()` (méthode *mailbox.Maildir*), 999
- `__setstate__()` (*copy protocol*), 407
- `__setstate__()` (méthode *object*), 403
- `__slots__`, 1763
- `__stderr__` (dans le module *sys*), 1552
- `__stdin__` (dans le module *sys*), 1552
- `__stdout__` (dans le module *sys*), 1552
- `__str__()` (méthode *datetime.date*), 175
- `__str__()` (méthode *datetime.datetime*), 183
- `__str__()` (méthode *datetime.time*), 187
- `__str__()` (méthode *email.charset.Charset*), 980
- `__str__()` (méthode *email.header.Header*), 978
- `__str__()` (méthode *email.headerregistry.Address*), 957
- `__str__()` (méthode *email.headerregistry.Group*), 958
- `__str__()` (méthode *email.message.EmailMessage*), 933
- `__str__()` (méthode *email.message.Message*), 968
- `__str__()` (méthode *multiprocessing.managers.BaseProxy*), 752
- `__sub__()` (dans le module *operator*), 344
- `__subclasses__()` (méthode *class*), 80
- `__subclasshook__()` (méthode *abc.ABCMeta*), 1576
- `__suppress_context__` (attribut *traceback.TracebackException*), 1583
- `__truediv__()` (dans le module *operator*), 344
- `__xor__()` (dans le module *operator*), 344
- `__anonymous__` (attribut *ctypes.Structure*), 712
- `__asdict__()` (méthode *collections.somenamedtuple*), 213
- `__b_base__` (attribut *ctypes._CData*), 709
- `__b_needsfree__` (attribut *ctypes._CData*), 709
- `__callmethod__` (méthode *multiprocessing.managers.BaseProxy*), 751
- `__CData` (classe dans *ctypes*), 708
- `__clear_type_cache__()` (dans le module *sys*), 1540
- `__current_frames__()` (dans le module *sys*), 1540
- `__debugmallocstats__()` (dans le module *sys*), 1540
- `__dummy_thread` (module), 796
- `__enablelegacywindowsfsencoding__()` (dans le module *sys*), 1552
- `__exit__()` (dans le module *os*), 551
- `__fields__` (attribut *ast.AST*), 1643
- `__fields__` (attribut *collections.somenamedtuple*), 214
- `__fields__` (attribut *ctypes.Structure*), 712
- `__flush__()` (méthode *wsgiref.handlers.BaseHandler*), 1102
- `__FuncPtr` (classe dans *ctypes*), 703
- `__get_child_mock__()` (méthode *unittest.mock.Mock*), 1419

- `_getframe()` (dans le module `sys`), 1545
 - `_getvalue()` (méthode `multiprocessing.managers.BaseProxy`), 752
 - `_handle` (attribut `ctypes.PyDLL`), 702
 - `_length_` (attribut `ctypes.Array`), 713
 - `_locale` module, 1251
 - `_make()` (méthode de la classe `collections.somenamedtuple`), 213
 - `_makeResult()` (méthode `unittest.TextTestRunner`), 1409
 - `_name` (attribut `ctypes.PyDLL`), 702
 - `_objects` (attribut `ctypes._CData`), 709
 - `_pack_` (attribut `ctypes.Structure`), 712
 - `_parse()` (méthode `gettext.NullTranslations`), 1246
 - `_Pointer` (classe dans `ctypes`), 713
 - `_replace()` (méthode `collections.somenamedtuple`), 213
 - `_setroot()` (méthode `xml.etree.ElementTree.ElementTree`), 1041
 - `_SimpleCData` (classe dans `ctypes`), 709
 - `_source` (attribut `collections.somenamedtuple`), 213
 - `_structure()` (dans le module `email.iterators`), 984
 - `_thread` (module), 794
 - `_type_` (attribut `ctypes._Pointer`), 713
 - `_type_` (attribut `ctypes.Array`), 713
 - `_write()` (méthode `wsgiref.handlers.BaseHandler`), 1102
 - `_xoptions` (dans le module `sys`), 1554
 - `{ }` (curly brackets)
 - in regular expressions, 103
 - in string formatting, 93
 - `|` (vertical bar)
 - in regular expressions, 104
 - opérateur, 30
 - `~` (tilde)
 - home directory expansion, 368
 - opérateur, 30
- ## A
- `-a`
 - `pickletools` command line option, 1675
 - `A` (dans le module `re`), 108
 - `a2b_base64()` (dans le module `binascii`), 1019
 - `a2b_hex()` (dans le module `binascii`), 1020
 - `a2b_hqx()` (dans le module `binascii`), 1019
 - `a2b_qp()` (dans le module `binascii`), 1019
 - `a2b_uu()` (dans le module `binascii`), 1019
 - `a85decode()` (dans le module `base64`), 1016
 - `a85encode()` (dans le module `base64`), 1016
 - `ABC` (classe dans `abc`), 1575
 - `abc` (module), 1575
 - `ABCMeta` (classe dans `abc`), 1575
 - `abiflags` (dans le module `sys`), 1539
 - `abort()` (dans le module `os`), 550
 - `abort()` (méthode `asyncio.DatagramTransport`), 885
 - `abort()` (méthode `asyncio.WriteTransport`), 884
 - `abort()` (méthode `ftplib.FTP`), 1141
 - `abort()` (méthode `threading.Barrier`), 726
 - `above()` (méthode `curses.panel.Panel`), 671
 - `abs()` (dans le module `operator`), 343
 - `abs()` (fonction de base), 5
 - `abs()` (méthode `decimal.Context`), 294
 - `abspath()` (dans le module `os.path`), 368
 - `AbstractBasicAuthHandler` (classe dans `urllib.request`), 1108
 - `abstractclassmethod()` (dans le module `abc`), 1578
 - `AbstractContextManager` (classe dans `contextlib`), 1563
 - `AbstractDigestAuthHandler` (classe dans `urllib.request`), 1108
 - `AbstractEventLoop` (classe dans `asyncio`), 858
 - `AbstractEventLoopPolicy` (classe dans `asyncio`), 872
 - `AbstractFormatter` (classe dans `formatter`), 1679
 - `abstractmethod()` (dans le module `abc`), 1577
 - `abstractproperty()` (dans le module `abc`), 1578
 - `AbstractSet` (classe dans `typing`), 1355
 - `abstractstaticmethod()` (dans le module `abc`), 1578
 - `AbstractWriter` (classe dans `formatter`), 1681
 - `accept()` (méthode `asyncore.dispatcher`), 917
 - `accept()` (méthode `multiprocessing.connection.Listener`), 755
 - `accept()` (méthode `socket.socket`), 807
 - `access()` (dans le module `os`), 533
 - `accumulate()` (dans le module `itertools`), 323
 - `acos()` (dans le module `cmath`), 279
 - `acos()` (dans le module `math`), 275
 - `acosh()` (dans le module `cmath`), 279
 - `acosh()` (dans le module `math`), 276
 - `acquire()` (méthode `_thread.lock`), 795
 - `acquire()` (méthode `asyncio.Condition`), 905
 - `acquire()` (méthode `asyncio.Lock`), 905
 - `acquire()` (méthode `asyncio.Semaphore`), 906
 - `acquire()` (méthode `logging.Handler`), 618
 - `acquire()` (méthode `multiprocessing.Lock`), 742
 - `acquire()` (méthode `multiprocessing.RLock`), 742
 - `acquire()` (méthode `threading.Condition`), 721
 - `acquire()` (méthode `threading.Lock`), 719
 - `acquire()` (méthode `threading.RLock`), 720
 - `acquire()` (méthode `threading.Semaphore`), 723
 - `acquire_lock()` (dans le module `imp`), 1747
 - `action` (attribut `optparse.Option`), 1732
 - `Action` (classe dans `argparse`), 600
 - `ACTIONS` (attribut `optparse.Option`), 1743

- `active_children()` (dans le module `multiprocessing`), 738
- `active_count()` (dans le module `threading`), 715
- `add()` (dans le module `audioop`), 1223
- `add()` (dans le module `operator`), 343
- `add()` (méthode `decimal.Context`), 294
- `add()` (méthode `frozenset`), 72
- `add()` (méthode `mailbox.Mailbox`), 996
- `add()` (méthode `mailbox.Maildir`), 999
- `add()` (méthode `msilib.RadioButtonGroup`), 1688
- `add()` (méthode `pstats.Stats`), 1499
- `add()` (méthode `tarfile.TarFile`), 465
- `add()` (méthode `tkinter.ttk.Notebook`), 1320
- `add_alias()` (dans le module `email.charset`), 980
- `add_alternative()` (méthode `email.message.EmailMessage`), 939
- `add_argument()` (méthode `argparse.ArgumentParser`), 591
- `add_argument_group()` (méthode `argparse.ArgumentParser`), 608
- `add_attachment()` (méthode `email.message.EmailMessage`), 939
- `add_cgi_vars()` (méthode `wsgiref.handlers.BaseHandler`), 1102
- `add_charset()` (dans le module `email.charset`), 980
- `add_codec()` (dans le module `email.charset`), 981
- `add_cookie_header()` (méthode `http.cookiejar.CookieJar`), 1191
- `add_data()` (dans le module `msilib`), 1684
- `add_done_callback()` (méthode `asyncio.Future`), 876
- `add_done_callback()` (méthode `concurrent.futures.Future`), 772
- `add_fallback()` (méthode `gettext.NullTranslations`), 1246
- `add_file()` (méthode `msilib.Directory`), 1687
- `add_flag()` (méthode `mailbox.MaildirMessage`), 1004
- `add_flag()` (méthode `mailbox.mboxMessage`), 1006
- `add_flag()` (méthode `mailbox.MMDFMessage`), 1010
- `add_flow_data()` (méthode `formatter.formatter`), 1678
- `add_folder()` (méthode `mailbox.Maildir`), 999
- `add_folder()` (méthode `mailbox.MH`), 1001
- `add_get_handler()` (méthode `email.contentmanager.ContentManager`), 958
- `add_handler()` (méthode `urllib.request.OpenerDirector`), 1110
- `add_header()` (méthode `email.message.EmailMessage`), 935
- `add_header()` (méthode `email.message.Message`), 970
- `add_header()` (méthode `urllib.request.Request`), 1110
- `add_header()` (méthode `wsgiref.headers.Headers`), 1098
- `add_history()` (dans le module `readline`), 141
- `add_hor_rule()` (méthode `formatter.formatter`), 1678
- `add_label()` (méthode `mailbox.BabylMessage`), 1008
- `add_label_data()` (méthode `formatter.formatter`), 1678
- `add_line_break()` (méthode `formatter.formatter`), 1678
- `add_literal_data()` (méthode `formatter.formatter`), 1678
- `add_mutually_exclusive_group()` (méthode `argparse.ArgumentParser`), 608
- `add_option()` (méthode `optparse.OptionParser`), 1730
- `add_parent()` (méthode `urllib.request.BaseHandler`), 1111
- `add_password()` (méthode `urllib.request.HTTPPasswordMgr`), 1113
- `add_password()` (méthode `urllib.request.HTTPPasswordMgrWithPriorAuth`), 1114
- `add_reader()` (méthode `asyncio.AbstractEventLoop`), 863
- `add_related()` (méthode `email.message.EmailMessage`), 938
- `add_section()` (méthode `configparser.ConfigParser`), 491
- `add_section()` (méthode `configparser.RawConfigParser`), 493
- `add_sequence()` (méthode `mailbox.MHMessage`), 1007
- `add_set_handler()` (méthode `email.contentmanager.ContentManager`), 958
- `add_signal_handler()` (méthode `asyncio.AbstractEventLoop`), 865
- `add_stream()` (dans le module `msilib`), 1684
- `add_subparsers()` (méthode `argparse.ArgumentParser`), 604
- `add_tables()` (dans le module `msilib`), 1684
- `add_type()` (dans le module `mimetypes`), 1013
- `add_unredirected_header()` (méthode `urllib.request.Request`), 1110
- `add_writer()` (méthode `asyncio.AbstractEventLoop`), 863
- `addch()` (méthode `curses.window`), 657
- `addCleanup()` (méthode `unittest.TestCase`), 1403
- `addcomponent()` (méthode `turtle.Shape`), 1288
- `addError()` (méthode `unittest.TestResult`), 1408
- `addExpectedFailure()` (méthode `unittest.TestResult`), 1408
- `addFailure()` (méthode `unittest.TestResult`), 1408
- `addfile()` (méthode `tarfile.TarFile`), 465
- `addFilter()` (méthode `logging.Handler`), 618
- `addFilter()` (méthode `logging.Logger`), 617
- `addHandler()` (méthode `logging.Logger`), 617
- `addLevelName()` (dans le module `logging`), 626
- `addnstr()` (méthode `curses.window`), 657

- `AddPackagePath()` (dans le module `modulefinder`), 1619
- `addr` (attribut `smtplib.SMTPChannel`), 1167
- `addr_spec` (attribut `email.headerregistry.Address`), 957
- `address` (attribut `email.headerregistry.SingleAddressHeader`), 955
- `address` (attribut `multiprocessing.connection.Listener`), 756
- `address` (attribut `multiprocessing.managers.BaseManager`), 747
- `Address` (classe dans `email.headerregistry`), 957
- `address_exclude()` (méthode `ipaddress.IPv4Network`), 1216
- `address_exclude()` (méthode `ipaddress.IPv6Network`), 1218
- `address_family` (attribut `socketserver.BaseServer`), 1176
- `address_string()` (méthode `http.server.BaseHTTPRequestHandler`), 1184
- `addresses` (attribut `email.headerregistry.AddressHeader`), 955
- `addresses` (attribut `email.headerregistry.Group`), 957
- `AddressHeader` (classe dans `email.headerregistry`), 955
- `addressof()` (dans le module `ctypes`), 706
- `AddressValueError`, 1222
- `addshape()` (dans le module `turtle`), 1286
- `addsitedir()` (dans le module `site`), 1607
- `addSkip()` (méthode `unittest.TestResult`), 1408
- `addstr()` (méthode `curses.window`), 657
- `addSubTest()` (méthode `unittest.TestResult`), 1409
- `addSuccess()` (méthode `unittest.TestResult`), 1408
- `addTest()` (méthode `unittest.TestSuite`), 1404
- `addTests()` (méthode `unittest.TestSuite`), 1404
- `addTypeEqualityFunc()` (méthode `unittest.TestCase`), 1401
- `addUnexpectedSuccess()` (méthode `unittest.TestResult`), 1409
- `adjusted()` (méthode `decimal.Decimal`), 287
- `adler32()` (dans le module `zlib`), 439
- `ADPCM`, Intel/DVI, 1223
- `adpcm2lin()` (dans le module `audioop`), 1223
- `AF_ALG` (dans le module `socket`), 802
- `AF_CAN` (dans le module `socket`), 801
- `AF_INET` (dans le module `socket`), 800
- `AF_INET6` (dans le module `socket`), 800
- `AF_LINK` (dans le module `socket`), 802
- `AF_PACKET` (dans le module `socket`), 801
- `AF_RDS` (dans le module `socket`), 801
- `AF_UNIX` (dans le module `socket`), 800
- `aifc` (module), 1226
- `aifc()` (méthode `aifc.aifc`), 1228
- `AIFF`, 1226, 1234
- `aiff()` (méthode `aifc.aifc`), 1228
- `AIFF-C`, 1226, 1234
- `alarm()` (dans le module `signal`), 923
- `A-LAW`, 1228, 1237
- `a-LAW`, 1223
- `alaw2lin()` (dans le module `audioop`), 1223
- `ALERT_DESCRIPTION_HANDSHAKE_FAILURE` (dans le module `ssl`), 827
- `ALERT_DESCRIPTION_INTERNAL_ERROR` (dans le module `ssl`), 827
- `AlertDescription` (classe dans `ssl`), 828
- `algorithms_available` (dans le module `hashlib`), 505
- `algorithms_guaranteed` (dans le module `hashlib`), 505
- `alias` (`pdb` command), 1494
- `alias` de type, 1764
- `alignment()` (dans le module `ctypes`), 706
- `alive` (attribut `weakref.finalize`), 234
- `all()` (fonction de base), 5
- `all_errors` (dans le module `ftplib`), 1140
- `all_features` (dans le module `xml.sax.handler`), 1065
- `all_frames` (attribut `tracemalloc.Filter`), 1517
- `all_properties` (dans le module `xml.sax.handler`), 1066
- `all_suffixes()` (dans le module `importlib.machinery`), 1630
- `all_tasks()` (méthode de la classe `asyncio.Task`), 878
- `allocate_lock()` (dans le module `_thread`), 794
- `allow_reuse_address` (attribut `socketserver.BaseServer`), 1176
- `allowed_domains()` (méthode `http.cookiejar.DefaultCookiePolicy`), 1195
- `alt()` (dans le module `curses.ascii`), 670
- `ALT_DIGITS` (dans le module `locale`), 1255
- `altsep` (dans le module `os`), 561
- `altzone` (dans le module `time`), 581
- `ALWAYS_TYPED_ACTIONS` (attribut `optparse.Option`), 1744
- `AMPER` (dans le module `token`), 1651
- `AMPEREQUAL` (dans le module `token`), 1651
- `and`
opérateur, 27, 28
- `and_()` (dans le module `operator`), 343
- `--annotate`
`pickletools` command line option, 1675
- `annotation`, 1753
- `annotation` (attribut `inspect.Parameter`), 1597
- `annotation` de fonction, 1757
- `annotation` de variable, 1764
- `answer_challenge()` (dans le module `multiprocessing.connection`), 755
- `anticipate_failure()` (dans le module `test.support`), 1478
- `Any` (dans le module `typing`), 1360

- ANY (dans le module `unittest.mock`), 1442
 any() (fonction de base), 5
 AnyStr (dans le module `typing`), 1362
 API provisoire, 1762
 api_version (dans le module `sys`), 1553
 apop() (méthode `poplib.POP3`), 1144
 append() (méthode `array.array`), 229
 append() (méthode `collections.deque`), 207
 append() (méthode `email.header.Header`), 978
 append() (méthode `imaplib.IMAP4`), 1148
 append() (méthode `msilib.CAB`), 1686
 append() (méthode `pipes.Template`), 1712
 append() (méthode `xml.etree.ElementTree.Element`), 1040
 append() (sequence method), 37
 append_history_file() (dans le module `readline`), 140
 appendChild() (méthode `xml.dom.Node`), 1049
 appendleft() (méthode `collections.deque`), 207
 application_uri() (dans le module `wsgiref.util`), 1096
 apply (2to3 fixer), 1469
 apply() (méthode `multiprocessing.pool.Pool`), 753
 apply_async() (méthode `multiprocessing.pool.Pool`), 753
 apply_defaults() (méthode `inspect.BoundArguments`), 1598
 architecture() (dans le module `platform`), 672
 archive (attribut `zipimport.zipimporter`), 1616
 aRepr (dans le module `reprlib`), 249
 argparse (module), 581
 args (attribut `BaseException`), 82
 args (attribut `functools.partial`), 342
 args (attribut `inspect.BoundArguments`), 1598
 args (attribut `subprocess.CompletedProcess`), 775
 args (attribut `subprocess.Popen`), 781
 args (pdb command), 1494
 argtypes (attribut `ctypes._FuncPtr`), 703
 argument, 1753
 argument nommé, 1759
 argument positionnel, 1762
 ArgumentDefaultsHelpFormatter (classe dans `argparse`), 587
 ArgumentError, 704
 ArgumentParser (classe dans `argparse`), 583
 arguments (attribut `inspect.BoundArguments`), 1598
 argv (dans le module `sys`), 1539
 arithmetic, 29
 ArithmeticError, 82
 array
 module, 51
 array (classe dans `array`), 229
 Array (classe dans `ctypes`), 713
 array (module), 228
 Array() (dans le module `multiprocessing`), 744
 Array() (dans le module `multiprocessing.sharedctypes`), 744
 Array() (méthode `multiprocessing.managers.SyncManager`), 748
 arrays, 228
 arraysize (attribut `sqlite3.Cursor`), 430
 arrêt de l'interpréteur, 1758
 article() (méthode `nnplib.NNTP`), 1157
 as_bytes() (méthode `email.message.EmailMessage`), 933
 as_bytes() (méthode `email.message.Message`), 968
 as_completed() (dans le module `asyncio`), 880
 as_completed() (dans le module `concurrent.futures`), 773
 as_integer_ratio() (méthode `decimal.Decimal`), 287
 as_integer_ratio() (méthode `float`), 32
 AS_IS (dans le module `formatter`), 1678
 as_posix() (méthode `pathlib.PurePath`), 358
 as_string() (méthode `email.message.EmailMessage`), 933
 as_string() (méthode `email.message.Message`), 967
 as_tuple() (méthode `decimal.Decimal`), 287
 as_uri() (méthode `pathlib.PurePath`), 358
 ASCII (dans le module `re`), 108
 ascii() (dans le module `curses.ascii`), 670
 ascii() (fonction de base), 6
 ascii_letters (dans le module `string`), 91
 ascii_lowercase (dans le module `string`), 91
 ascii_uppercase (dans le module `string`), 91
 asctime() (dans le module `time`), 574
 asin() (dans le module `cmath`), 279
 asin() (dans le module `math`), 275
 asinh() (dans le module `cmath`), 279
 asinh() (dans le module `math`), 276
 assert
 état, 83
 assert_any_call() (méthode `unittest.mock.Mock`), 1418
 assert_called() (méthode `unittest.mock.Mock`), 1417
 assert_called_once() (méthode `unittest.mock.Mock`), 1417
 assert_called_once_with() (méthode `unittest.mock.Mock`), 1417
 assert_called_with() (méthode `unittest.mock.Mock`), 1417
 assert_has_calls() (méthode `unittest.mock.Mock`), 1418
 assert_line_data() (méthode `formatter.formatter`), 1679
 assert_not_called() (méthode `unittest.mock.Mock`), 1418

- `assertAlmostEqual()` (méthode `unittest.TestCase`), 1400
`assertCountEqual()` (méthode `unittest.TestCase`), 1401
`assertDictEqual()` (méthode `unittest.TestCase`), 1402
`assertEqual()` (méthode `unittest.TestCase`), 1397
`assertFalse()` (méthode `unittest.TestCase`), 1397
`assertGreater()` (méthode `unittest.TestCase`), 1400
`assertGreaterEqual()` (méthode `unittest.TestCase`), 1400
`assertIn()` (méthode `unittest.TestCase`), 1397
`AssertionError`, 83
`assertIs()` (méthode `unittest.TestCase`), 1397
`assertIsInstance()` (méthode `unittest.TestCase`), 1397
`assertIsNone()` (méthode `unittest.TestCase`), 1397
`assertIsNot()` (méthode `unittest.TestCase`), 1397
`assertIsNotNone()` (méthode `unittest.TestCase`), 1397
`assertLess()` (méthode `unittest.TestCase`), 1400
`assertLessEqual()` (méthode `unittest.TestCase`), 1400
`assertListEqual()` (méthode `unittest.TestCase`), 1402
`assertLogs()` (méthode `unittest.TestCase`), 1399
`assertMultiLineEqual()` (méthode `unittest.TestCase`), 1401
`assertNotAlmostEqual()` (méthode `unittest.TestCase`), 1400
`assertNotEqual()` (méthode `unittest.TestCase`), 1397
`assertNotIn()` (méthode `unittest.TestCase`), 1397
`assertNotIsInstance()` (méthode `unittest.TestCase`), 1397
`assertNotRegex()` (méthode `unittest.TestCase`), 1401
`assertRaises()` (méthode `unittest.TestCase`), 1398
`assertRaisesRegex()` (méthode `unittest.TestCase`), 1398
`assertRegex()` (méthode `unittest.TestCase`), 1401
`asserts` (2to3 fixer), 1469
`assertSequenceEqual()` (méthode `unittest.TestCase`), 1402
`assertSetEqual()` (méthode `unittest.TestCase`), 1402
`assertTrue()` (méthode `unittest.TestCase`), 1397
`assertTupleEqual()` (méthode `unittest.TestCase`), 1402
`assertWarns()` (méthode `unittest.TestCase`), 1399
`assertWarnsRegex()` (méthode `unittest.TestCase`), 1399
`assignment`
 `slice`, 37
 `subscript`, 37
`AST` (classe dans `ast`), 1643
`ast` (module), 1643
`astimezone()` (méthode `datetime.datetime`), 180
`ASYNC` (dans le module `token`), 1651
`async()` (dans le module `asyncio`), 880
`async_chat` (classe dans `asynchat`), 919
`async_chat.ac_in_buffer_size` (dans le module `asynchat`), 919
`async_chat.ac_out_buffer_size` (dans le module `asynchat`), 919
`AsyncContextManager` (classe dans `typing`), 1357
`AsyncGenerator` (classe dans `collections.abc`), 221
`AsyncGenerator` (classe dans `typing`), 1357
`AsyncGeneratorType` (dans le module `types`), 240
`asynchat` (module), 919
`asyncio` (module), 857
`asyncio.subprocess.DEVNULL` (dans le module `asyncio`), 900
`asyncio.subprocess.PIPE` (dans le module `asyncio`), 900
`asyncio.subprocess.Process` (classe dans `asyncio`), 900
`asyncio.subprocess.STDOUT` (dans le module `asyncio`), 900
`AsyncIterable` (classe dans `collections.abc`), 221
`AsyncIterable` (classe dans `typing`), 1356
`AsyncIterator` (classe dans `collections.abc`), 221
`AsyncIterator` (classe dans `typing`), 1356
`asyncore` (module), 915
`AsyncResult` (classe dans `multiprocessing.pool`), 754
`AT` (dans le module `token`), 1651
`at_eof()` (méthode `asyncio.StreamReader`), 894
`atan()` (dans le module `cmath`), 279
`atan()` (dans le module `math`), 275
`atan2()` (dans le module `math`), 275
`atanh()` (dans le module `cmath`), 279
`atanh()` (dans le module `math`), 276
`ATEQUAL` (dans le module `token`), 1651
`atexit` (attribut `weakref.finalize`), 234
`atexit` (module), 1579
`atof()` (dans le module `locale`), 1256
`atoi()` (dans le module `locale`), 1257
`attach()` (méthode `email.message.Message`), 968
`attach_mock()` (méthode `unittest.mock.Mock`), 1419
`AttlistDeclHandler()` (méthode `xml.parsers.expat.xmlparser`), 1077
`attrgetter()` (dans le module `operator`), 345
`attrib` (attribut `xml.etree.ElementTree.Element`), 1039
`attribut`, 1754
`AttributeError`, 83
`attributes` (attribut `xml.dom.Node`), 1048
`AttributesImpl` (classe dans `xml.sax.xmlreader`), 1070
`AttributesNSImpl` (classe dans `xml.sax.xmlreader`), 1070
`attroff()` (méthode `curses.window`), 657

[attron\(\)](#) (méthode `curses.window`), 657
[attrset\(\)](#) (méthode `curses.window`), 657
[Audio Interchange File Format](#), 1226, 1234
[AUDIO_FILE_ENCODING_ADPCM_G721](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_ADPCM_G722](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_ADPCM_G723_3](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_ADPCM_G723_5](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_ALAW_8](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_DOUBLE](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_FLOAT](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_LINEAR_8](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_LINEAR_16](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_LINEAR_24](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_LINEAR_32](#) (dans le module `sunau`), 1229
[AUDIO_FILE_ENCODING_MULAW_8](#) (dans le module `sunau`), 1229
[AUDIO_FILE_MAGIC](#) (dans le module `sunau`), 1229
[AUDIODEV](#), 1238
[audioop](#) (module), 1223
[auth\(\)](#) (méthode `ftplib.FTP_TLS`), 1143
[auth\(\)](#) (méthode `smtplib.SMTP`), 1162
[authenticate\(\)](#) (méthode `imaplib.IMAP4`), 1148
[AuthenticationError](#), 735
[authenticators\(\)](#) (méthode `netrc.netrc`), 495
[authkey](#) (attribut `multiprocessing.Process`), 734
[auto](#) (classe dans `enum`), 251
[autorange\(\)](#) (méthode `timeit.Timer`), 1504
[avg\(\)](#) (dans le module `audioop`), 1224
[avgpp\(\)](#) (dans le module `audioop`), 1224
[avoids_symlink_attacks](#) (attribut `shutil.rmtree`), 391
[AWAIT](#) (dans le module `token`), 1651
[awaitable](#), 1754
[Awaitable](#) (classe dans `collections.abc`), 220
[Awaitable](#) (classe dans `typing`), 1356

B

-b
 [compileall](#) command line option, 1660
 [unittest](#) command line option, 1389
[b2a_base64\(\)](#) (dans le module `binascii`), 1019
[b2a_hex\(\)](#) (dans le module `binascii`), 1020
[b2a_hqx\(\)](#) (dans le module `binascii`), 1020

[b2a_qp\(\)](#) (dans le module `binascii`), 1019
[b2a_uu\(\)](#) (dans le module `binascii`), 1019
[b16decode\(\)](#) (dans le module `base64`), 1016
[b16encode\(\)](#) (dans le module `base64`), 1016
[b32decode\(\)](#) (dans le module `base64`), 1016
[b32encode\(\)](#) (dans le module `base64`), 1016
[b64decode\(\)](#) (dans le module `base64`), 1015
[b64encode\(\)](#) (dans le module `base64`), 1015
[b85decode\(\)](#) (dans le module `base64`), 1017
[b85encode\(\)](#) (dans le module `base64`), 1017
[Babyl](#) (classe dans `mailbox`), 1002
[BabylMessage](#) (classe dans `mailbox`), 1008
[back\(\)](#) (dans le module `turtle`), 1266
[backslashreplace_errors\(\)](#) (dans le module `codecs`), 155
[backward\(\)](#) (dans le module `turtle`), 1266
[BadStatusLine](#), 1134
[BadZipFile](#), 453
[BadZipfile](#), 453
[Balloon](#) (classe dans `tkinter.tix`), 1332
[Barrier](#) (classe dans `multiprocessing`), 741
[Barrier](#) (classe dans `threading`), 725
[Barrier\(\)](#) (méthode `multiprocessing.managers.SyncManager`), 747
[base64](#)
 [encoding](#), 1015
 [module](#), 1019
[base64](#) (module), 1015
[base_exec_prefix](#) (dans le module `sys`), 1539
[base_prefix](#) (dans le module `sys`), 1539
[BaseCGIHandler](#) (classe dans `wsgiref.handlers`), 1101
[BaseCookie](#) (classe dans `http.cookies`), 1186
[BaseEventLoop](#) (classe dans `asyncio`), 858
[BaseException](#), 82
[BaseHandler](#) (classe dans `urllib.request`), 1107
[BaseHandler](#) (classe dans `wsgiref.handlers`), 1101
[BaseHeader](#) (classe dans `email.headerregistry`), 953
[BaseHTTPRequestHandler](#) (classe dans `http.server`), 1182
[BaseManager](#) (classe dans `multiprocessing.managers`), 746
[basename\(\)](#) (dans le module `os.path`), 368
[BaseProxy](#) (classe dans `multiprocessing.managers`), 751
[BaseRequestHandler](#) (classe dans `socketserver`), 1177
[BaseRotatingHandler](#) (classe dans `logging.handlers`), 640
[BaseSelector](#) (classe dans `selectors`), 855
[BaseServer](#) (classe dans `socketserver`), 1175
[basestring](#) (2to3 fixer), 1470
[BaseSubprocessTransport](#) (classe dans `asyncio`), 885
[BaseTransport](#) (classe dans `asyncio`), 883
[basicConfig\(\)](#) (dans le module `logging`), 627

- BasicContext (*classe dans decimal*), 292
- BasicInterpolation (*classe dans configparser*), 482
- baudrate() (*dans le module curses*), 651
- bbox() (*méthode tkinter.ttk.Treeview*), 1325
- BDADDR_ANY (*dans le module socket*), 802
- BDADDR_LOCAL (*dans le module socket*), 802
- bdb
 - module, 1489
- Bdb (*classe dans bdb*), 1484
- bdb (*module*), 1483
- BdbQuit, 1483
- BDFL, 1754
- beep() (*dans le module curses*), 651
- Beep() (*dans le module winsound*), 1698
- BEFORE_ASYNC_WITH (*opcode*), 1668
- begin_fill() (*dans le module turtle*), 1275
- begin_poly() (*dans le module turtle*), 1280
- below() (*méthode curses.panel.Panel*), 671
- Benchmarking, 1503
- benchmarking, 574
- betavariate() (*dans le module random*), 311
- bgcolor() (*dans le module turtle*), 1281
- bgpic() (*dans le module turtle*), 1281
- bias() (*dans le module audioop*), 1224
- bidirectional() (*dans le module unicodedata*), 136
- BigEndianStructure (*classe dans ctypes*), 711
- bin() (*fonction de base*), 6
- binary
 - data, packing, 145
 - literals, 29
- Binary (*classe dans msilib*), 1684
- Binary (*classe dans xmlrpc.client*), 1201
- binary mode, 17
- binary semaphores, 794
- BINARY_ADD (*opcode*), 1667
- BINARY_AND (*opcode*), 1667
- BINARY_FLOOR_DIVIDE (*opcode*), 1666
- BINARY_LSHIFT (*opcode*), 1667
- BINARY_MATRIX_MULTIPLY (*opcode*), 1666
- BINARY_MODULO (*opcode*), 1667
- BINARY_MULTIPLY (*opcode*), 1666
- BINARY_OR (*opcode*), 1667
- BINARY_POWER (*opcode*), 1666
- BINARY_RSHIFT (*opcode*), 1667
- BINARY_SUBSCR (*opcode*), 1667
- BINARY_SUBTRACT (*opcode*), 1667
- BINARY_TRUE_DIVIDE (*opcode*), 1666
- BINARY_XOR (*opcode*), 1667
- BinaryIO (*classe dans typing*), 1358
- binascii (*module*), 1019
- bind (widgets), 1312
- bind() (*méthode asyncio.dispatcher*), 917
- bind() (*méthode inspect.Signature*), 1596
- bind() (*méthode socket.socket*), 807
- bind_partial() (*méthode inspect.Signature*), 1596
- bind_port() (*dans le module test.support*), 1479
- bind_textdomain_codeset() (*dans le module gettext*), 1244
- bindtextdomain() (*dans le module gettext*), 1243
- bindtextdomain() (*dans le module locale*), 1258
- binhex
 - module, 1019
- binhex (*module*), 1018
- binhex() (*dans le module binhex*), 1018
- bisect (*module*), 226
- bisect() (*dans le module bisect*), 226
- bisect_left() (*dans le module bisect*), 226
- bisect_right() (*dans le module bisect*), 226
- bit_length() (*méthode int*), 31
- bitmap() (*méthode msilib.Dialog*), 1688
- bitwise
 - operations, 30
- bk() (*dans le module turtle*), 1266
- bkgd() (*méthode curses.window*), 657
- bkgdset() (*méthode curses.window*), 658
- blake2b() (*dans le module hashlib*), 507
- blake2b, blake2s, 507
- blake2b.MAX_DIGEST_SIZE (*dans le module hashlib*), 508
- blake2b.MAX_KEY_SIZE (*dans le module hashlib*), 508
- blake2b.PERSON_SIZE (*dans le module hashlib*), 508
- blake2b.SALT_SIZE (*dans le module hashlib*), 508
- blake2s() (*dans le module hashlib*), 507
- blake2s.MAX_DIGEST_SIZE (*dans le module hashlib*), 508
- blake2s.MAX_KEY_SIZE (*dans le module hashlib*), 508
- blake2s.PERSON_SIZE (*dans le module hashlib*), 508
- blake2s.SALT_SIZE (*dans le module hashlib*), 508
- block_size (attribut hmac.HMAC), 514
- blocked_domains() (*méthode http.cookiejar.DefaultCookiePolicy*), 1195
- BlockingIOError, 87, 564
- body() (*méthode nntplib.NNTP*), 1157
- body_encode() (*méthode email.charset.Charset*), 980
- body_encoding (attribut email.charset.Charset), 979
- body_line_iterator() (*dans le module email.iterators*), 984
- BOM (*dans le module codecs*), 153
- BOM_BE (*dans le module codecs*), 153
- BOM_LE (*dans le module codecs*), 153
- BOM_UTF8 (*dans le module codecs*), 153
- BOM_UTF16 (*dans le module codecs*), 153
- BOM_UTF16_BE (*dans le module codecs*), 153
- BOM_UTF16_LE (*dans le module codecs*), 153
- BOM_UTF32 (*dans le module codecs*), 153
- BOM_UTF32_BE (*dans le module codecs*), 153

- BOM_UTF32_LE (*dans le module codecs*), 153
- bool (*classe de base*), 6
- Boolean
 - objet, 29
 - operations, 27, 28
 - type, 6
 - values, 79
- BOOLEAN_STATES (*attribut configparser.ConfigParser*), 487
- bootstrap() (*dans le module ensurepip*), 1523
- border() (*méthode curses.window*), 658
- bottom() (*méthode curses.panel.Panel*), 671
- bottom_panel() (*dans le module curses.panel*), 671
- BoundArguments (*classe dans inspect*), 1598
- BoundaryError, 952
- BoundedSemaphore (*classe dans asyncio*), 907
- BoundedSemaphore (*classe dans multiprocessing*), 741
- BoundedSemaphore (*classe dans threading*), 723
- BoundedSemaphore() (*méthode multiprocessing.managers.SyncManager*), 747
- box() (*méthode curses.window*), 658
- bpformat() (*méthode bdb.Breakpoint*), 1484
- bpprint() (*méthode bdb.Breakpoint*), 1484
- break (*pdb command*), 1492
- break_anywhere() (*méthode bdb.Bdb*), 1485
- break_here() (*méthode bdb.Bdb*), 1485
- break_long_words (*attribut textwrap.TextWrapper*), 135
- BREAK_LOOP (*opcode*), 1668
- break_on_hyphens (*attribut textwrap.TextWrapper*), 135
- Breakpoint (*classe dans bdb*), 1483
- breakpoints, 1339
- broadcast_address (*attribut dress.IPv4Network*), 1216
- broadcast_address (*attribut dress.IPv6Network*), 1218
- broken (*attribut threading.Barrier*), 726
- BrokenBarrierError, 726
- BrokenPipeError, 87
- BrokenProcessPool, 773
- BROWSER, 1085, 1086
- BsdDbShelf (*classe dans shelve*), 412
- buffer
 - unittest command line option, 1389
- buffer (2to3 fixer), 1470
- buffer (*attribut io.TextIOBase*), 570
- buffer (*attribut unittest.TestResult*), 1407
- buffer protocol
 - binary sequence types, 51
 - str (*built-in class*), 42
- buffer size, I/O, 17
- buffer_info() (*méthode array.array*), 230
- buffer_size (*attribut xml.parsers.expat.xmlparser*), 1076
- buffer_text (*attribut xml.parsers.expat.xmlparser*), 1076
- buffer_used (*attribut xml.parsers.expat.xmlparser*), 1076
- BufferedIOBase (*classe dans io*), 567
- BufferedRandom (*classe dans io*), 570
- BufferedReader (*classe dans io*), 569
- BufferedRWPair (*classe dans io*), 570
- BufferedWriter (*classe dans io*), 569
- BufferError, 82
- BufferingHandler (*classe dans logging.handlers*), 647
- BufferTooShort, 735
- bufsize() (*méthode ossaudiodev.oss_audio_device*), 1240
- BUILD_CONST_KEY_MAP (*opcode*), 1670
- BUILD_LIST (*opcode*), 1670
- BUILD_LIST_UNPACK (*opcode*), 1671
- BUILD_MAP (*opcode*), 1670
- BUILD_MAP_UNPACK (*opcode*), 1671
- BUILD_MAP_UNPACK_WITH_CALL (*opcode*), 1671
- build_opener() (*dans le module urllib.request*), 1105
- BUILD_SET (*opcode*), 1670
- BUILD_SET_UNPACK (*opcode*), 1671
- BUILD_SLICE (*opcode*), 1673
- BUILD_STRING (*opcode*), 1671
- BUILD_TUPLE (*opcode*), 1670
- BUILD_TUPLE_UNPACK (*opcode*), 1671
- BUILD_TUPLE_UNPACK_WITH_CALL (*opcode*), 1671
- built-in
 - types, 27
- builtin_module_names (*dans le module sys*), 1540
- BuiltinFunctionType (*dans le module types*), 240
- BuiltinImporter (*classe dans importlib.machinery*), 1630
- BuiltinMethodType (*dans le module types*), 240
- builtins (*module*), 1557
- ButtonBox (*classe dans tkinter.tix*), 1332
- bye() (*dans le module turtle*), 1287
- byref() (*dans le module ctypes*), 706
- bytearray
 - formatting, 62
 - interpolation, 62
 - methods, 53
 - objet, 37, 51, 52
- bytearray (*classe de base*), 52
- byte-code
 - file, 1659, 1745
- Bytecode (*classe dans dis*), 1663
- BYTECODE_SUFFIXES (*dans le module importlib.machinery*), 1630
- Bytecode.codeobj (*dans le module dis*), 1663

Bytecode.first_line (*dans le module dis*), 1663
 byteorder (*dans le module sys*), 1540
 bytes
 formatting, 62
 interpolation, 62
 methods, 53
 objet, 51
 str (*built-in class*), 42
 bytes (*attribut uuid.UUID*), 1171
 bytes (*classe de base*), 51
 bytes_le (*attribut uuid.UUID*), 1171
 BytesFeedParser (*classe dans email.parser*), 940
 BytesGenerator (*classe dans email.generator*), 943
 BytesHeaderParser (*classe dans email.parser*), 941
 BytesIO (*classe dans io*), 569
 BytesParser (*classe dans email.parser*), 941
 ByteString (*classe dans collections.abc*), 220
 ByteString (*classe dans typing*), 1355
 byteswap() (*dans le module audioop*), 1224
 byteswap() (*méthode array.array*), 230
 BytesWarning, 89
 bz2 (*module*), 445
 BZ2Compressor (*classe dans bz2*), 447
 BZ2Decompressor (*classe dans bz2*), 447
 BZ2File (*classe dans bz2*), 446

C

C
 language, 29, 30
 structures, 145
 -C
 trace command line option, 1509
 -c
 timeit command line option, 1506
 trace command line option, 1508
 unittest command line option, 1389
 -c <tarfile> <source1> ... <sourceN>
 tarfile command line option, 468
 -c <zipfile> <source1> ... <sourceN>
 zipfile command line option, 460
 c_bool (*classe dans ctypes*), 711
 C_BUILTIN (*dans le module imp*), 1748
 c_byte (*classe dans ctypes*), 709
 c_char (*classe dans ctypes*), 709
 c_char_p (*classe dans ctypes*), 709
 c_contiguous (*attribut memoryview*), 70
 c_double (*classe dans ctypes*), 710
 C_EXTENSION (*dans le module imp*), 1748
 c_float (*classe dans ctypes*), 710
 c_int (*classe dans ctypes*), 710
 c_int8 (*classe dans ctypes*), 710
 c_int16 (*classe dans ctypes*), 710
 c_int32 (*classe dans ctypes*), 710
 c_int64 (*classe dans ctypes*), 710
 c_long (*classe dans ctypes*), 710
 c_longdouble (*classe dans ctypes*), 710
 c_longlong (*classe dans ctypes*), 710
 c_short (*classe dans ctypes*), 710
 c_size_t (*classe dans ctypes*), 710
 c_ssize_t (*classe dans ctypes*), 710
 c_ubyte (*classe dans ctypes*), 710
 c_uint (*classe dans ctypes*), 710
 c_uint8 (*classe dans ctypes*), 710
 c_uint16 (*classe dans ctypes*), 710
 c_uint32 (*classe dans ctypes*), 711
 c_uint64 (*classe dans ctypes*), 711
 c_ulong (*classe dans ctypes*), 711
 c_ulonglong (*classe dans ctypes*), 711
 c_ushort (*classe dans ctypes*), 711
 c_void_p (*classe dans ctypes*), 711
 c_wchar (*classe dans ctypes*), 711
 c_wchar_p (*classe dans ctypes*), 711
 CAB (*classe dans msilib*), 1686
 cache_from_source() (*dans le module imp*), 1746
 cache_from_source() (*dans le module importlib.util*), 1634
 cached (*attribut importlib.machinery.ModuleSpec*), 1633
 CacheFTPHandler (*classe dans urllib.request*), 1109
 calcsiz() (*dans le module struct*), 146
 Calendar (*classe dans calendar*), 199
 calendar (*module*), 199
 calendar() (*dans le module calendar*), 201
 call() (*dans le module subprocess*), 783
 call() (*dans le module unittest.mock*), 1440
 call_args (*attribut unittest.mock.Mock*), 1421
 call_args_list (*attribut unittest.mock.Mock*), 1422
 call_at() (*méthode asyncio.AbstractEventLoop*), 859
 call_count (*attribut unittest.mock.Mock*), 1420
 call_exception_handler() (*méthode asyncio.AbstractEventLoop*), 866
 CALL_FUNCTION (*opcode*), 1673
 CALL_FUNCTION_EX (*opcode*), 1673
 CALL_FUNCTION_KW (*opcode*), 1673
 call_later() (*méthode asyncio.AbstractEventLoop*), 859
 call_list() (*méthode unittest.mock.call*), 1440
 call_soon() (*méthode asyncio.AbstractEventLoop*), 859
 call_soon_threadsafe() (*méthode asyncio.AbstractEventLoop*), 859
 call_tracing() (*dans le module sys*), 1540
 Callable (*classe dans collections.abc*), 219
 Callable (*dans le module typing*), 1361
 callable() (*fonction de base*), 7
 CallableProxyType (*dans le module weakref*), 235
 callback (*attribut optparse.Option*), 1732
 callback() (*méthode contextlib.ExitStack*), 1568
 callback_args (*attribut optparse.Option*), 1732

- `callback_kwargs` (attribut *optparse.Option*), 1732
- `callbacks` (dans le module *gc*), 1590
- `called` (attribut *unittest.mock.Mock*), 1420
- `CalledProcessError`, 776
- `CAN_BCM` (dans le module *socket*), 801
- `can_change_color()` (dans le module *curses*), 651
- `can_fetch()` (méthode *url-lib.robotparser.RobotFileParser*), 1129
- `CAN_RAW_FD_FRAMES` (dans le module *socket*), 801
- `can_symlink()` (dans le module *test.support*), 1478
- `can_write_eof()` (méthode *asyncio.StreamWriter*), 894
- `can_write_eof()` (méthode *asyncio.WriteTransport*), 884
- `cancel()` (méthode *asyncio.Future*), 876
- `cancel()` (méthode *asyncio.Handle*), 867
- `cancel()` (méthode *asyncio.Task*), 878
- `cancel()` (méthode *concurrent.futures.Future*), 772
- `cancel()` (méthode *sched.scheduler*), 790
- `cancel()` (méthode *threading.Timer*), 725
- `cancel_dump_traceback_later()` (dans le module *faulthandler*), 1488
- `cancel_join_thread()` (méthode *multiprocessing.Queue*), 737
- `cancelled()` (méthode *asyncio.Future*), 876
- `cancelled()` (méthode *concurrent.futures.Future*), 772
- `CancelledError`, 773
- `CannotSendHeader`, 1134
- `CannotSendRequest`, 1134
- `canonic()` (méthode *bdb.Bdb*), 1484
- `canonical()` (méthode *decimal.Context*), 294
- `canonical()` (méthode *decimal.Decimal*), 287
- `capa()` (méthode *poplib.POP3*), 1144
- `capitalize()` (méthode *bytearray*), 58
- `capitalize()` (méthode *bytes*), 58
- `capitalize()` (méthode *str*), 42
- `captured_stderr()` (dans le module *test.support*), 1477
- `captured_stdin()` (dans le module *test.support*), 1477
- `captured_stdout()` (dans le module *test.support*), 1477
- `captureWarnings()` (dans le module *logging*), 628
- `capwords()` (dans le module *string*), 102
- `casefold()` (méthode *str*), 42
- `cast()` (dans le module *ctypes*), 706
- `cast()` (dans le module *typing*), 1359
- `cast()` (méthode *memoryview*), 67
- `cat()` (dans le module *nis*), 1716
- `--catch`
 - `unittest` command line option, 1389
- `catch_warnings` (classe dans *warnings*), 1563
- `category()` (dans le module *unicodedata*), 136
- `cbreak()` (dans le module *curses*), 651
- `ccc()` (méthode *ftplib.FTP_TLS*), 1143
- `C-contiguous`, 1755
- `CDLL` (classe dans *ctypes*), 701
- `ceil()` (dans le module *math*), 273
- `ceil()` (in module *math*), 30
- `center()` (méthode *bytearray*), 56
- `center()` (méthode *bytes*), 56
- `center()` (méthode *str*), 42
- `CERT_NONE` (dans le module *ssl*), 823
- `CERT_OPTIONAL` (dans le module *ssl*), 823
- `CERT_REQUIRED` (dans le module *ssl*), 824
- `cert_store_stats()` (méthode *ssl.SSLContext*), 832
- `cert_time_to_seconds()` (dans le module *ssl*), 822
- `CertificateError`, 818
- `certificates`, 837
- `CFUNCTYPE()` (dans le module *ctypes*), 704
- `CGI`
 - `debugging`, 1093
 - `exceptions`, 1095
 - `protocol`, 1088
 - `security`, 1092
 - `tracebacks`, 1095
- `cgi` (module), 1088
- `cgi_directories` (attribut *http.server.CGIHTTPRequestHandler*), 1186
- `CGIHandler` (classe dans *wsgiref.handlers*), 1101
- `CGIHTTPRequestHandler` (classe dans *http.server*), 1185
- `cgitb` (module), 1095
- `CGIXMLRPCRequestHandler` (classe dans *xmlrpc.server*), 1206
- `chain()` (dans le module *itertools*), 324
- `chaîne entre triple guillemets`, 1763
- `chaining`
 - `comparisons`, 28
- `ChainMap` (classe dans *collections*), 202
- `ChainMap` (classe dans *typing*), 1357
- `change_cwd()` (dans le module *test.support*), 1478
- `CHANNEL_BINDING_TYPES` (dans le module *ssl*), 827
- `channel_class` (attribut *smtpd.SMTPServer*), 1166
- `channels()` (méthode *ossaudiodev.oss_audio_device*), 1239
- `CHAR_MAX` (dans le module *locale*), 1257
- `character`, 136
- `CharacterDataHandler()` (méthode *xml.parsers.expat.xmlparser*), 1077
- `characters()` (méthode *xml.sax.handler.ContentHandler*), 1067
- `characters_written` (attribut *BlockingIOError*), 87
- `chargeur`, 1759
- `Charset` (classe dans *email.charset*), 979
- `charset()` (méthode *gettext.NullTranslations*), 1246
- `chdir()` (dans le module *os*), 534
- `check` (attribut *lzma.LZMADecompressor*), 451

- `check()` (dans le module `tabnanny`), 1657
- `check()` (méthode `imaplib.IMAP4`), 1148
- `check_all__()` (dans le module `test.support`), 1480
- `check_call()` (dans le module `subprocess`), 783
- `check_hostname` (attribut `ssl.SSLContext`), 836
- `check_output()` (dans le module `subprocess`), 784
- `check_output()` (méthode `doctest.OutputChecker`), 1382
- `check_returncode()` (méthode `subprocess.CompletedProcess`), 775
- `check_unused_args()` (méthode `string.Formatter`), 93
- `check_warnings()` (dans le module `test.support`), 1477
- `checkbox()` (méthode `msilib.Dialog`), 1688
- `checkcache()` (dans le module `linecache`), 388
- `checkfuncname()` (dans le module `bdb`), 1486
- `CheckList` (classe dans `tkinter.tix`), 1333
- `checksum`
 - Cyclic Redundancy Check, 440
- chemin des importations, 1758
- chercheur, 1757
- chercheur basé sur les chemins, 1761
- chercheur dans les méta-chemins, 1760
- chercheur de chemins, 1761
- `chflags()` (dans le module `os`), 534
- `chgat()` (méthode `curses.window`), 658
- `childNodes` (attribut `xml.dom.Node`), 1048
- `ChildProcessError`, 87
- `chmod()` (dans le module `os`), 535
- `chmod()` (méthode `pathlib.Path`), 362
- `choice()` (dans le module `random`), 310
- `choice()` (dans le module `secrets`), 515
- `choices` (attribut `optparse.Option`), 1732
- `choices()` (dans le module `random`), 310
- `chown()` (dans le module `os`), 535
- `chown()` (dans le module `shutil`), 391
- `chr()` (fonction de base), 7
- `chroot()` (dans le module `os`), 536
- `Chunk` (classe dans `chunk`), 1234
- `chunk` (module), 1234
- `cipher`
 - DES, 1705
- `cipher()` (méthode `ssl.SSLSocket`), 830
- `circle()` (dans le module `turtle`), 1268
- `CIRCUMFLEX` (dans le module `token`), 1651
- `CIRCUMFLEXEQUAL` (dans le module `token`), 1651
- `Clamped` (classe dans `decimal`), 298
- `Class` (classe dans `syntable`), 1650
- `Class browser`, 1336
- classe, 1755
- classe de base abstraite, 1753
- `classmethod()` (fonction de base), 7
- `ClassVar` (dans le module `typing`), 1361
- `CLD_CONTINUED` (dans le module `os`), 556
- `CLD_DUMPED` (dans le module `os`), 556
- `CLD_EXITED` (dans le module `os`), 556
- `CLD_TRAPPED` (dans le module `os`), 556
- `clean()` (méthode `mailbox.Maildir`), 999
- `cleandoc()` (dans le module `inspect`), 1595
- `clear` (*pdb command*), 1492
- `Clear Breakpoint`, 1339
- `clear()` (dans le module `turtle`), 1275, 1282
- `clear()` (méthode `asyncio.Event`), 905
- `clear()` (méthode `collections.deque`), 207
- `clear()` (méthode `curses.window`), 658
- `clear()` (méthode `dict`), 74
- `clear()` (méthode `email.message.EmailMessage`), 939
- `clear()` (méthode `frozenset`), 72
- `clear()` (méthode `http.cookiejar.CookieJar`), 1192
- `clear()` (méthode `mailbox.Mailbox`), 998
- `clear()` (méthode `threading.Event`), 724
- `clear()` (méthode `xml.etree.ElementTree.Element`), 1039
- `clear()` (sequence method), 37
- `clear_all_breaks()` (méthode `bdb.Bdb`), 1486
- `clear_all_file_breaks()` (méthode `bdb.Bdb`), 1486
- `clear_bppynumber()` (méthode `bdb.Bdb`), 1486
- `clear_break()` (méthode `bdb.Bdb`), 1486
- `clear_cache()` (dans le module `filecmp`), 379
- `clear_content()` (méthode `email.message.EmailMessage`), 939
- `clear_flags()` (méthode `decimal.Context`), 293
- `clear_frames()` (dans le module `traceback`), 1582
- `clear_history()` (dans le module `readline`), 141
- `clear_session_cookies()` (méthode `http.cookiejar.CookieJar`), 1192
- `clear_traces()` (dans le module `tracemalloc`), 1515
- `clear_traps()` (méthode `decimal.Context`), 293
- `clearcache()` (dans le module `linecache`), 387
- `ClearData()` (méthode `msilib.Record`), 1686
- `clearok()` (méthode `curses.window`), 658
- `clearscreen()` (dans le module `turtle`), 1282
- `clearstamp()` (dans le module `turtle`), 1269
- `clearstamps()` (dans le module `turtle`), 1269
- `Client()` (dans le module `multiprocessing.connection`), 755
- `client_address` (attribut `http.server.BaseHTTPRequestHandler`), 1182
- `--clock`
 - timeit command line option, 1506
- `clock()` (dans le module `time`), 574
- `clock_getres()` (dans le module `time`), 575
- `clock_gettime()` (dans le module `time`), 575
- `CLOCK_HIGHRES` (dans le module `time`), 580
- `CLOCK_MONOTONIC` (dans le module `time`), 580
- `CLOCK_MONOTONIC_RAW` (dans le module `time`), 580

- CLOCK_PROCESS_CPUTIME_ID (dans le module *time*), 580
- CLOCK_REALTIME (dans le module *time*), 580
- clock_settime() (dans le module *time*), 575
- CLOCK_THREAD_CPUTIME_ID (dans le module *time*), 580
- clone() (dans le module *turtle*), 1280
- clone() (méthode *email.generator.BytesGenerator*), 944
- clone() (méthode *email.generator.Generator*), 945
- clone() (méthode *email.policy.Policy*), 948
- clone() (méthode *pipes.Template*), 1712
- cloneNode() (méthode *xml.dom.Node*), 1049
- close() (dans le module *fileinput*), 373
- close() (dans le module *os*), 526
- close() (méthode *aifc.aifc*), 1227, 1228
- close() (méthode *asyncio.AbstractEventLoop*), 858
- close() (méthode *asyncio.BaseSubprocessTransport*), 885
- close() (méthode *asyncio.BaseTransport*), 883
- close() (méthode *asyncio.Server*), 867
- close() (méthode *asyncio.StreamWriter*), 894
- close() (méthode *asyncore.dispatcher*), 917
- close() (méthode *chunk.Chunk*), 1234
- close() (méthode *contextlib.ExitStack*), 1568
- close() (méthode *dbm.dumb.dumbdbm*), 418
- close() (méthode *dbm.gnu.gdbm*), 417
- close() (méthode *dbm.ndbm.ndbm*), 418
- close() (méthode *email.parser.BytesFeedParser*), 941
- close() (méthode *ftplib.FTP*), 1143
- close() (méthode *html.parser.HTMLParser*), 1025
- close() (méthode *http.client.HTTPConnection*), 1135
- close() (méthode *imaplib.IMAP4*), 1148
- close() (méthode *io.IOBase*), 565
- close() (méthode *logging.FileHandler*), 639
- close() (méthode *logging.Handler*), 619
- close() (méthode *logging.handlers.MemoryHandler*), 647
- close() (méthode *logging.handlers.NTEventLogHandler*), 646
- close() (méthode *logging.handlers.SocketHandler*), 643
- close() (méthode *logging.handlers.SysLogHandler*), 644
- close() (méthode *mailbox.Mailbox*), 998
- close() (méthode *mailbox.Maildir*), 999
- close() (méthode *mailbox.MH*), 1002
- close() (méthode *mmap.mmap*), 928
- Close() (méthode *msilib.View*), 1685
- close() (méthode *multiprocessing.connection.Connection*), 740
- close() (méthode *multiprocessing.connection.Listener*), 756
- close() (méthode *multiprocessing.pool.Pool*), 754
- close() (méthode *multiprocessing.Queue*), 737
- close() (méthode *ossaudiodev.oss_audio_device*), 1238
- close() (méthode *ossaudiodev.oss_mixer_device*), 1240
- close() (méthode *os.scandir*), 540
- close() (méthode *select.devpoll*), 849
- close() (méthode *select.epoll*), 850
- close() (méthode *select.kqueue*), 852
- close() (méthode *selectors.BaseSelector*), 855
- close() (méthode *shelve.Shelf*), 411
- close() (méthode *socket.socket*), 807
- close() (méthode *sqlite3.Connection*), 423
- close() (méthode *sqlite3.Cursor*), 429
- close() (méthode *sunau.AU_read*), 1230
- close() (méthode *sunau.AU_write*), 1231
- close() (méthode *tarfile.TarFile*), 465
- close() (méthode *telnetlib.Telnet*), 1169
- close() (méthode *urllib.request.BaseHandler*), 1111
- close() (méthode *wave.Wave_read*), 1232
- close() (méthode *wave.Wave_write*), 1233
- Close() (méthode *winreg.PyHKEY*), 1697
- close() (méthode *xml.etree.ElementTree.TreeBuilder*), 1043
- close() (méthode *xml.etree.ElementTree.XMLParser*), 1043
- close() (méthode *xml.etree.ElementTree.XMLPullParser*), 1044
- close() (méthode *xml.sax.xmlreader.IncrementalParser*), 1072
- close() (méthode *zipfile.ZipFile*), 455
- close_connection (attribut *http.server.BaseHTTPRequestHandler*), 1182
- close_when_done() (méthode *asynchat.async_chat*), 919
- closed (attribut *http.client.HTTPResponse*), 1137
- closed (attribut *io.IOBase*), 565
- closed (attribut *mmap.mmap*), 928
- closed (attribut *ossaudiodev.oss_audio_device*), 1240
- closed (attribut *select.devpoll*), 849
- closed (attribut *select.epoll*), 850
- closed (attribut *select.kqueue*), 852
- CloseKey() (dans le module *winreg*), 1691
- closelog() (dans le module *syslog*), 1717
- closerange() (dans le module *os*), 526
- closing() (dans le module *contextlib*), 1564
- clrtoebot() (méthode *curses.window*), 658
- clrtoeol() (méthode *curses.window*), 658
- cmath (module), 278
- cmd
 - module, 1489
- cmd (attribut *subprocess.CalledProcessError*), 776
- cmd (attribut *subprocess.TimeoutExpired*), 775
- Cmd (classe dans *cmd*), 1293
- cmd (module), 1293
- cmdloop() (méthode *cmd.Cmd*), 1293
- cmdqueue (attribut *cmd.Cmd*), 1295
- cmp() (dans le module *filecmp*), 379
- cmp_op (dans le module *dis*), 1674

- `cmp_to_key()` (dans le module `functools`), 336
`cmpfiles()` (dans le module `filecmp`), 379
`CMSG_LEN()` (dans le module `socket`), 805
`CMSG_SPACE()` (dans le module `socket`), 806
`CO_ASYNC_GENERATOR` (dans le module `inspect`), 1605
`CO_COROUTINE` (dans le module `inspect`), 1605
`CO_GENERATOR` (dans le module `inspect`), 1604
`CO_ITERABLE_COROUTINE` (dans le module `inspect`), 1605
`CO_NESTED` (dans le module `inspect`), 1604
`CO_NEWLOCALS` (dans le module `inspect`), 1604
`CO_NOFREE` (dans le module `inspect`), 1604
`CO_OPTIMIZED` (dans le module `inspect`), 1604
`CO_VARARGS` (dans le module `inspect`), 1604
`CO_VARKEYWORDS` (dans le module `inspect`), 1604
`code` (attribut `SystemExit`), 86
`code` (attribut `urllib.error.HTTPError`), 1129
`code` (attribut `xml.etree.ElementTree.ParseError`), 1045
`code` (attribut `xml.parsers.expat.ExpatError`), 1079
`code` (module), 1611
`code` intermédiaire (bytecode), 1755
`code` object, 78, 414
`code_info()` (dans le module `dis`), 1664
`CodecInfo` (classe dans `codecs`), 151
`Codecs`, 151
 `decode`, 151
 `encode`, 151
`codecs` (module), 151
`coded_value` (attribut `http.cookies.Morsel`), 1188
`codeop` (module), 1613
`codepoint2name` (dans le module `html.entities`), 1028
`codes` (dans le module `xml.parsers.expat.errors`), 1081
`CODESET` (dans le module locale), 1254
`CodeType` (dans le module `types`), 240
`coercition`, 1755
`col_offset` (attribut `ast.AST`), 1644
`collapse_addresses()` (dans le module `ipaddress`), 1221
`collapse_rfc2231_value()` (dans le module `email.utils`), 984
`collect()` (dans le module `gc`), 1588
`collect_incoming_data()` (méthode `asyncio.chat.async_chat`), 919
`Collection` (classe dans `collections.abc`), 220
`Collection` (classe dans `typing`), 1355
`collections` (module), 202
`collections.abc` (module), 218
`colno` (attribut `json.JSONDecodeError`), 991
`colno` (attribut `re.error`), 113
`COLON` (dans le module `token`), 1651
`color()` (dans le module `turtle`), 1274
`color_content()` (dans le module `curses`), 651
`color_pair()` (dans le module `curses`), 651
`colormode()` (dans le module `turtle`), 1285
`colorsys` (module), 1235
`COLS`, 656
`column()` (méthode `tkinter.ttk.Treeview`), 1325
`COLUMNS`, 657
`columns` (attribut `os.terminal_size`), 532
`combinations()` (dans le module `itertools`), 324
`combinations_with_replacement()` (dans le module `itertools`), 325
`combine()` (méthode de la classe `datetime.datetime`), 178
`combining()` (dans le module `unicodedata`), 136
`ComboBox` (classe dans `tkinter.tix`), 1332
`Combobox` (classe dans `tkinter.ttk`), 1319
`COMMA` (dans le module `token`), 1651
`command` (attribut `http.server.BaseHTTPRequestHandler`), 1182
`CommandCompiler` (classe dans `codeop`), 1614
`commands` (`pdb` `command`), 1493
`comment` (attribut `http.cookiejar.Cookie`), 1196
`comment` (attribut `zipfile.ZipFile`), 457
`comment` (attribut `zipfile.ZipInfo`), 459
`COMMENT` (dans le module `tokenize`), 1653
`Comment()` (dans le module `xml.etree.ElementTree`), 1037
`comment_url` (attribut `http.cookiejar.Cookie`), 1196
`commenters` (attribut `shlex.shlex`), 1300
`CommentHandler()` (méthode `xml.parsers.expat.xmlparser`), 1078
`commit()` (méthode `msilib.CAB`), 1686
`Commit()` (méthode `msilib.Database`), 1684
`commit()` (méthode `sqlite3.Connection`), 422
`common` (attribut `filecmp.dircmp`), 380
`Common Gateway Interface`, 1088
`common_dirs` (attribut `filecmp.dircmp`), 380
`common_files` (attribut `filecmp.dircmp`), 380
`common_funny` (attribut `filecmp.dircmp`), 380
`common_types` (dans le module `mimetypes`), 1013
`commonpath()` (dans le module `os.path`), 368
`commonprefix()` (dans le module `os.path`), 368
`communicate()` (méthode `asyncio.asyncio.subprocess.Process`), 900
`communicate()` (méthode `subprocess.Popen`), 781
`compare()` (méthode `decimal.Context`), 294
`compare()` (méthode `decimal.Decimal`), 287
`compare()` (méthode `difflib.Differ`), 129
`compare_digest()` (dans le module `hmac`), 515
`compare_digest()` (dans le module `secrets`), 517
`compare_networks()` (méthode `ipaddress.IPv4Network`), 1217
`compare_networks()` (méthode `ipaddress.IPv6Network`), 1218
`COMPARE_OP` (opcode), 1671
`compare_signal()` (méthode `decimal.Context`), 294
`compare_signal()` (méthode `decimal.Decimal`), 287
`compare_to()` (méthode `tracemalloc.Snapshot`), 1517

- `compare_total()` (méthode `decimal.Context`), 294
- `compare_total()` (méthode `decimal.Decimal`), 287
- `compare_total_mag()` (méthode `decimal.Context`), 294
- `compare_total_mag()` (méthode `decimal.Decimal`), 288
- `comparing`
 - objects, 28
- `comparison`
 - operator, 28
- `COMPARISON_FLAGS` (dans le module `doctest`), 1372
- `comparisons`
 - chaining, 28
- `Compat32` (classe dans `email.policy`), 951
- `compat32` (dans le module `email.policy`), 952
- `compile`
 - fonction de base, 78, 240, 1641
- `Compile` (classe dans `codeop`), 1614
- `compile()` (dans le module `py_compile`), 1659
- `compile()` (dans le module `re`), 108
- `compile()` (fonction de base), 7
- `compile()` (méthode `parser.ST`), 1642
- `compile_command()` (dans le module `code`), 1612
- `compile_command()` (dans le module `codeop`), 1613
- `compile_dir()` (dans le module `compileall`), 1661
- `compile_file()` (dans le module `compileall`), 1661
- `compile_path()` (dans le module `compileall`), 1662
- `compileall` (module), 1660
- `compileall` command line option
 - b, 1660
 - d `destdir`, 1660
 - directory ..., 1660
 - f, 1660
 - file ..., 1660
 - i list, 1660
 - j N, 1660
 - l, 1660
 - q, 1660
 - r, 1660
 - x `regex`, 1660
- `compilest()` (dans le module `parser`), 1641
- `complete()` (méthode `rlcompleter.Completer`), 144
- `complete_statement()` (dans le module `sqlite3`), 421
- `completedefault()` (méthode `cmd.Cmd`), 1294
- `CompletedProcess` (classe dans `subprocess`), 775
- `complex`
 - fonction de base, 29
- `Complex` (classe dans `numbers`), 269
- `complex` (classe de base), 8
- `complex number`
 - literals, 29
 - objet, 29
- `compress()` (dans le module `bz2`), 448
- `compress()` (dans le module `gzip`), 444
- `compress()` (dans le module `itertools`), 326
- `compress()` (dans le module `lzma`), 451
- `compress()` (dans le module `zlib`), 439
- `compress()` (méthode `bz2.BZ2Compressor`), 447
- `compress()` (méthode `lzma.LZMACompressor`), 450
- `compress()` (méthode `zlib.Compress`), 441
- `compress_size` (attribut `zipfile.ZipInfo`), 460
- `compress_type` (attribut `zipfile.ZipInfo`), 459
- `compressed` (attribut `ipaddress.IPv4Address`), 1212
- `compressed` (attribut `ipaddress.IPv4Network`), 1216
- `compressed` (attribut `ipaddress.IPv6Address`), 1213
- `compressed` (attribut `ipaddress.IPv6Network`), 1218
- `compression()` (méthode `ssl.SSLSocket`), 830
- `CompressionError`, 462
- `compressobj()` (dans le module `zlib`), 440
- `COMSPEC`, 555, 778
- `concat()` (dans le module `operator`), 344
- `concatenation`
 - operation, 35
- `concurrent.futures` (module), 768
- `Condition` (classe dans `asyncio`), 905
- `Condition` (classe dans `multiprocessing`), 741
- `Condition` (classe dans `threading`), 721
- `condition` (`pdb` command), 1493
- `condition()` (méthode `msilib.Control`), 1687
- `Condition()` (méthode `multiprocessing.managers.SyncManager`), 748
- `ConfigParser` (classe dans `configparser`), 490
- `configparser` (module), 478
- `configuration`
 - file, 478
 - file, debugger, 1492
 - file, path, 1606
- `configuration information`, 1554
- `configure()` (méthode `tkinter.ttk.Style`), 1327
- `configure_mock()` (méthode `unittest.mock.Mock`), 1419
- `confstr()` (dans le module `os`), 560
- `confstr_names` (dans le module `os`), 560
- `conjugate()` (complex number method), 29
- `conjugate()` (méthode `decimal.Decimal`), 288
- `conjugate()` (méthode `numbers.Complex`), 269
- `conn` (attribut `smtpd.SMTPChannel`), 1167
- `connect()` (dans le module `sqlite3`), 421
- `connect()` (méthode `asyncore.dispatcher`), 916
- `connect()` (méthode `ftplib.FTP`), 1140
- `connect()` (méthode `http.client.HTTPConnection`), 1135
- `connect()` (méthode `multiprocessing.managers.BaseManager`), 747
- `connect()` (méthode `smtpplib.SMTP`), 1161
- `connect()` (méthode `socket.socket`), 807
- `connect_accepted_socket()` (méthode `asyncio.BaseEventLoop`), 863

- [connect_ex\(\)](#) (méthode `socket.socket`), 807
[connect_read_pipe\(\)](#) (méthode `asyncio.AbstractEventLoop`), 865
[connect_write_pipe\(\)](#) (méthode `asyncio.AbstractEventLoop`), 865
[connection](#) (attribut `sqlite3.Cursor`), 430
[Connection](#) (classe dans `multiprocessing.connection`), 740
[Connection](#) (classe dans `sqlite3`), 422
[connection_lost\(\)](#) (méthode `asyncio.BaseProtocol`), 886
[connection_made\(\)](#) (méthode `asyncio.BaseProtocol`), 886
[ConnectionAbortedError](#), 87
[ConnectionError](#), 87
[ConnectionRefusedError](#), 87
[ConnectionResetError](#), 87
[ConnectRegistry\(\)](#) (dans le module `winreg`), 1691
[const](#) (attribut `optparse.Option`), 1732
[constructor\(\)](#) (dans le module `copyreg`), 410
[consumed](#) (attribut `asyncio.LimitOverrunError`), 895
[container](#)
 iteration over, 34
[Container](#) (classe dans `collections.abc`), 219
[Container](#) (classe dans `typing`), 1355
[contains\(\)](#) (dans le module `operator`), 344
[content type](#)
 MIME, 1012
[content_manager](#) (attribut `email.policy.EmailPolicy`), 950
[content_type](#) (attribut `email.headerregistry.ContentTypeHeader`), 956
[ContentDispositionHeader](#) (classe dans `email.headerregistry`), 956
[ContentHandler](#) (classe dans `xml.sax.handler`), 1064
[ContentManager](#) (classe dans `email.contentmanager`), 958
[contents](#) (attribut `ctypes._Pointer`), 713
[ContentTooShortError](#), 1129
[ContentTransferEncoding](#) (classe dans `email.headerregistry`), 956
[ContentTypeHeader](#) (classe dans `email.headerregistry`), 956
[context](#) (attribut `ssl.SSLSocket`), 831
[Context](#) (classe dans `decimal`), 293
[context management protocol](#), 76
[context manager](#), 76
[context_diff\(\)](#) (dans le module `difflib`), 123
[ContextDecorator](#) (classe dans `contextlib`), 1566
[contextlib](#) (module), 1563
[ContextManager](#) (classe dans `typing`), 1356
[contextmanager\(\)](#) (dans le module `contextlib`), 1563
[contigu](#), 1755
[contiguous](#) (attribut `memoryview`), 70
[continue](#) (`pdb` command), 1493
[CONTINUE_LOOP](#) (opcode), 1668
[Control](#) (classe dans `msilib`), 1687
[Control](#) (classe dans `tkinter.tix`), 1332
[control\(\)](#) (méthode `msilib.Dialog`), 1688
[control\(\)](#) (méthode `select.kqueue`), 852
[controlnames](#) (dans le module `curses.ascii`), 671
[controls\(\)](#) (méthode `ossaudiodev.oss_mixer_device`), 1240
[ConversionError](#), 498
[conversions](#)
 numeric, 30
[convert_arg_line_to_args\(\)](#) (méthode `argparse.ArgumentParser`), 610
[convert_field\(\)](#) (méthode `string.Formatter`), 93
[Cookie](#) (classe dans `http.cookiejar`), 1191
[CookieError](#), 1186
[cookiejar](#) (attribut `lib.request.HTTPCookieProcessor`), 1113
[CookieJar](#) (classe dans `http.cookiejar`), 1190
[CookiePolicy](#) (classe dans `http.cookiejar`), 1190
[Coordinated Universal Time](#), 574
[Copy](#), 1339
[copy](#)
 module, 410
 protocol, 403
[copy](#) (module), 242
[copy\(\)](#) (dans le module `copy`), 242
[copy\(\)](#) (dans le module `multiprocessing.sharedctypes`), 745
[copy\(\)](#) (dans le module `shutil`), 389
[copy\(\)](#) (méthode `collections.deque`), 207
[copy\(\)](#) (méthode `decimal.Context`), 293
[copy\(\)](#) (méthode `dict`), 74
[copy\(\)](#) (méthode `frozenset`), 71
[copy\(\)](#) (méthode `hashlib.hash`), 505
[copy\(\)](#) (méthode `hmac.HMAC`), 514
[copy\(\)](#) (méthode `http.cookies.Morsel`), 1188
[copy\(\)](#) (méthode `imaplib.IMAP4`), 1148
[copy\(\)](#) (méthode `pipes.Template`), 1712
[copy\(\)](#) (méthode `types.MappingProxyType`), 241
[copy\(\)](#) (méthode `zlib.Compress`), 441
[copy\(\)](#) (méthode `zlib.Decompress`), 442
[copy\(\)](#) (sequence method), 37
[copy2\(\)](#) (dans le module `shutil`), 389
[copy_abs\(\)](#) (méthode `decimal.Context`), 294
[copy_abs\(\)](#) (méthode `decimal.Decimal`), 288
[copy_decimal\(\)](#) (méthode `decimal.Context`), 293
[copy_location\(\)](#) (dans le module `ast`), 1647
[copy_negate\(\)](#) (méthode `decimal.Context`), 294
[copy_negate\(\)](#) (méthode `decimal.Decimal`), 288
[copy_sign\(\)](#) (méthode `decimal.Context`), 294
[copy_sign\(\)](#) (méthode `decimal.Decimal`), 288

- `copyfile()` (dans le module `shutil`), 388
- `copyfileobj()` (dans le module `shutil`), 388
- copying files, 388
- `copymode()` (dans le module `shutil`), 389
- `copyreg` (module), 410
- `copyright` (dans le module `sys`), 1540
- `copyright` (variable de base), 26
- `copysign()` (dans le module `math`), 273
- `copystat()` (dans le module `shutil`), 389
- `copytree()` (dans le module `shutil`), 390
- `coroutine`, 1755
- `Coroutine` (classe dans `collections.abc`), 220
- `Coroutine` (classe dans `typing`), 1356
- `coroutine()` (dans le module `asyncio`), 873
- `coroutine()` (dans le module `types`), 242
- `CoroutineType` (dans le module `types`), 240
- `cos()` (dans le module `cmath`), 279
- `cos()` (dans le module `math`), 275
- `cosh()` (dans le module `cmath`), 279
- `cosh()` (dans le module `math`), 276
- `--count`
 - trace command line option, 1508
- `count` (attribut `tracemalloc.Statistic`), 1518
- `count` (attribut `tracemalloc.StatisticDiff`), 1518
- `count()` (dans le module `itertools`), 326
- `count()` (méthode `array.array`), 230
- `count()` (méthode `bytearray`), 53
- `count()` (méthode `bytes`), 53
- `count()` (méthode `collections.deque`), 208
- `count()` (méthode `str`), 42
- `count()` (sequence method), 35
- `count_diff` (attribut `tracemalloc.StatisticDiff`), 1518
- `Counter` (classe dans `collections`), 205
- `Counter` (classe dans `typing`), 1357
- `countOf()` (dans le module `operator`), 344
- `countTestCases()` (méthode `unittest.TestCase`), 1402
- `countTestCases()` (méthode `unittest.TestSuite`), 1404
- `CoverageResults` (classe dans `trace`), 1510
- `--coverdir=<dir>`
 - trace command line option, 1509
- `cProfile` (module), 1498
- CPU time, 574
- `cpu_count()` (dans le module `multiprocessing`), 738
- `cpu_count()` (dans le module `os`), 560
- `CPython`, 1755
- `crawl_delay()` (méthode `lib.robotparser.RobotFileParser`), 1129
- `CRC` (attribut `zipfile.ZipInfo`), 460
- `crc32()` (dans le module `binascii`), 1020
- `crc32()` (dans le module `zlib`), 440
- `crc_hqx()` (dans le module `binascii`), 1020
- `--create <tarfile> <source1> ... <sourceN>`
 - tarfile command line option, 468
- `create()` (dans le module `venv`), 1527
- `create()` (méthode `imaplib.IMAP4`), 1148
- `create()` (méthode `venv.EnvBuilder`), 1526
- `create_aggregate()` (méthode `sqlite3.Connection`), 423
- `create_archive()` (dans le module `zipapp`), 1533
- `create_autospec()` (dans le module `unittest.mock`), 1442
- `create_collation()` (méthode `sqlite3.Connection`), 424
- `create_configuration()` (méthode `venv.EnvBuilder`), 1527
- `create_connection()` (dans le module `socket`), 803
- `create_connection()` (méthode `asyncio.AbstractEventLoop`), 860
- `create_datagram_endpoint()` (méthode `asyncio.AbstractEventLoop`), 861
- `create_decimal()` (méthode `decimal.Context`), 293
- `create_decimal_from_float()` (méthode `decimal.Context`), 294
- `create_default_context()` (dans le module `ssl`), 820
- `create_function()` (méthode `sqlite3.Connection`), 423
- `create_future()` (méthode `asyncio.AbstractEventLoop`), 860
- `create_module()` (méthode `importlib.abc.Loader`), 1626
- `create_module()` (méthode `importlib.machinery.ExtensionFileLoader`), 1633
- `CREATE_NEW_CONSOLE` (dans le module `subprocess`), 783
- `CREATE_NEW_PROCESS_GROUP` (dans le module `subprocess`), 783
- `create_server()` (méthode `asyncio.AbstractEventLoop`), 862
- `create_socket()` (méthode `asyncore.dispatcher`), 916
- `create_stats()` (méthode `profile.Profile`), 1498
- `create_string_buffer()` (dans le module `ctypes`), 706
- `create_subprocess_exec()` (dans le module `asyncio`), 898
- `create_subprocess_shell()` (dans le module `asyncio`), 898
- `create_system` (attribut `zipfile.ZipInfo`), 459
- `url- create_task()` (méthode `asyncio.AbstractEventLoop`), 860
- `create_unicode_buffer()` (dans le module `ctypes`), 707
- `create_unix_connection()` (méthode `asyncio.AbstractEventLoop`), 862
- `create_unix_server()` (méthode `asyncio.AbstractEventLoop`), 863
- `create_version` (attribut `zipfile.ZipInfo`), 459

- createAttribute() (méthode *xml.dom.Document*), 1051
- createAttributeNS() (méthode *xml.dom.Document*), 1051
- createComment() (méthode *xml.dom.Document*), 1050
- createDocument() (méthode *xml.dom.DOMImplementation*), 1047
- createDocumentType() (méthode *xml.dom.DOMImplementation*), 1047
- createElement() (méthode *xml.dom.Document*), 1050
- createElementNS() (méthode *xml.dom.Document*), 1050
- createfilehandler() (méthode *tkinter.Widget.tk*), 1314
- CreateKey() (dans le module *winreg*), 1691
- CreateKeyEx() (dans le module *winreg*), 1691
- createLock() (méthode *logging.Handler*), 618
- createLock() (méthode *logging.NullHandler*), 639
- createProcessingInstruction() (méthode *xml.dom.Document*), 1050
- CreateRecord() (dans le module *msilib*), 1684
- createSocket() (méthode *logging.handlers.SocketHandler*), 643
- createTextNode() (méthode *xml.dom.Document*), 1050
- credits (variable de base), 26
- critical() (dans le module *logging*), 626
- critical() (méthode *logging.Logger*), 617
- CRNCYSTR (dans le module locale), 1255
- cross() (dans le module *audioop*), 1224
- crypt
 module, 1702
- crypt (module), 1705
- crypt() (dans le module *crypt*), 1705
- crypt(3), 1705
- cryptography, 503
- csv, 471
- csv (module), 471
- cte (attribut *email.headerregistry.ContentTransferEncoding*), 956
- cte_type (attribut *email.policy.Policy*), 947
- ctermid() (dans le module *os*), 520
- ctime() (dans le module *time*), 575
- ctime() (méthode *datetime.date*), 175
- ctime() (méthode *datetime.datetime*), 183
- ctrl() (dans le module *curses.ascii*), 670
- CTRL_BREAK_EVENT (dans le module *signal*), 922
- CTRL_C_EVENT (dans le module *signal*), 922
- ctypes (module), 681
- curdir (dans le module *os*), 560
- currency() (dans le module locale), 1256
- current() (méthode *tkinter.ttk.Combobox*), 1319
- current_process() (dans le module *multiprocessing*), 738
- current_task() (méthode de la classe *asyncio.Task*), 878
- current_thread() (dans le module *threading*), 715
- CurrentByteIndex (attribut *xml.parsers.expat.xmlparser*), 1076
- CurrentColumnNumber (attribut *xml.parsers.expat.xmlparser*), 1076
- currentframe() (dans le module *inspect*), 1602
- CurrentLineNumber (attribut *xml.parsers.expat.xmlparser*), 1076
- curs_set() (dans le module *curses*), 651
- curses (module), 650
- curses.ascii (module), 669
- curses.panel (module), 671
- curses.textpad (module), 667
- Cursor (classe dans *sqlite3*), 427
- cursor() (méthode *sqlite3.Connection*), 422
- cursyncup() (méthode *curses.window*), 658
- Cut, 1339
- cwd() (méthode de la classe *pathlib.Path*), 362
- cwd() (méthode *ftplib.FTP*), 1142
- cycle() (dans le module *itertools*), 326
- Cyclic Redundancy Check, 440
- ## D
- d destdir
 compileall command line option, 1660
- D_FMT (dans le module locale), 1254
- D_T_FMT (dans le module locale), 1254
- daemon (attribut *multiprocessing.Process*), 734
- daemon (attribut *threading.Thread*), 718
- data
 packingbinary, 145
- tabular, 471
- data (attribut *collections.UserDict*), 217
- data (attribut *collections.UserList*), 217
- data (attribut *select.kevent*), 853
- data (attribut *selectors.SelectorKey*), 855
- data (attribut *urllib.request.Request*), 1109
- data (attribut *xml.dom.Comment*), 1053
- data (attribut *xml.dom.ProcessingInstruction*), 1053
- data (attribut *xml.dom.Text*), 1053
- data (attribut *xmlrpc.client.Binary*), 1201
- Data (classe dans *plistlib*), 500
- data() (méthode *xml.etree.ElementTree.TreeBuilder*), 1043
- data_open() (méthode *urllib.request.DataHandler*), 1115
- data_received() (méthode *asyncio.Protocol*), 887
- database
 Unicode, 136
- DatabaseError, 431

- databases, 418
- `datagram_received()` (méthode `asyncio.DatagramProtocol`), 887
- `DatagramHandler` (classe dans `logging.handlers`), 644
- `DatagramProtocol` (classe dans `asyncio`), 886
- `DatagramRequestHandler` (classe dans `socketserver`), 1177
- `DataHandler` (classe dans `urllib.request`), 1109
- `date` (classe dans `datetime`), 173
- `date()` (méthode `datetime.datetime`), 180
- `date()` (méthode `nntplib.NNTP`), 1158
- `date_time` (attribut `zipfile.ZipInfo`), 459
- `date_time_string()` (méthode `http.server.BaseHTTPRequestHandler`), 1184
- `DateHeader` (classe dans `email.headerregistry`), 954
- `datetime` (attribut `email.headerregistry.DateHeader`), 954
- `datetime` (classe dans `datetime`), 177
- `DateTime` (classe dans `xmlrpc.client`), 1201
- `datetime` (module), 169
- `day` (attribut `datetime.date`), 174
- `day` (attribut `datetime.datetime`), 179
- `day_abbr` (dans le module `calendar`), 201
- `day_name` (dans le module `calendar`), 201
- `daylight` (dans le module `time`), 581
- Daylight Saving Time, 574
- `DbfilenameShelf` (classe dans `shelve`), 412
- `dbm` (module), 415
- `dbm.dumb` (module), 418
- `dbm.gnu`
 - module, 412
- `dbm.gnu` (module), 416
- `dbm.ndbm`
 - module, 412
- `dbm.ndbm` (module), 417
- `dcgettext()` (dans le module `locale`), 1258
- `debug` (attribut `imaplib.IMAP4`), 1152
- `debug` (attribut `shlex.shlex`), 1301
- `debug` (attribut `zipfile.ZipFile`), 457
- `DEBUG` (dans le module `re`), 109
- `debug()` (dans le module `doctest`), 1384
- `debug()` (dans le module `logging`), 624
- `debug()` (méthode `logging.Logger`), 616
- `debug()` (méthode `pipes.Template`), 1712
- `debug()` (méthode `unittest.TestCase`), 1396
- `debug()` (méthode `unittest.TestSuite`), 1404
- `DEBUG_BYTECODE_SUFFIXES` (dans le module `importlib.machinery`), 1630
- `DEBUG_COLLECTABLE` (dans le module `gc`), 1590
- `DEBUG_LEAK` (dans le module `gc`), 1591
- `DEBUG_SAVEALL` (dans le module `gc`), 1591
- `debug_src()` (dans le module `doctest`), 1384
- `DEBUG_STATS` (dans le module `gc`), 1590
- `DEBUG_UNCOLLECTABLE` (dans le module `gc`), 1590
- debugger, 1338, 1545, 1550
 - configuration file, 1492
- debugging, 1489
 - CGI, 1093
- `DebuggingServer` (classe dans `smtpd`), 1166
- `debuglevel` (attribut `http.client.HTTPResponse`), 1137
- `DebugRunner` (classe dans `doctest`), 1384
- `Decimal` (classe dans `decimal`), 286
- `decimal` (module), 281
- `decimal()` (dans le module `unicodedata`), 136
- `DecimalException` (classe dans `decimal`), 298
- decode
 - Codecs, 151
- `decode` (attribut `codecs.CodecInfo`), 151
- `decode()` (dans le module `base64`), 1017
- `decode()` (dans le module `codecs`), 151
- `decode()` (dans le module `quopri`), 1021
- `decode()` (dans le module `uu`), 1022
- `decode()` (méthode `bytearray`), 53
- `decode()` (méthode `bytes`), 53
- `decode()` (méthode `codecs.Codec`), 156
- `decode()` (méthode `codecs.IncrementalDecoder`), 157
- `decode()` (méthode `json.JSONDecoder`), 990
- `decode()` (méthode `xmlrpc.client.Binary`), 1201
- `decode()` (méthode `xmlrpc.client.DateTime`), 1201
- `decode_header()` (dans le module `email.header`), 978
- `decode_header()` (dans le module `nntplib`), 1158
- `decode_params()` (dans le module `email.utils`), 984
- `decode_rfc2231()` (dans le module `email.utils`), 983
- `decode_source()` (dans le module `importlib.util`), 1634
- `decodebytes()` (dans le module `base64`), 1017
- `DecodedGenerator` (classe dans `email.generator`), 945
- `decodestring()` (dans le module `base64`), 1017
- `decodestring()` (dans le module `quopri`), 1021
- `decomposition()` (dans le module `unicodedata`), 136
- `decompress()` (dans le module `bz2`), 448
- `decompress()` (dans le module `gzip`), 444
- `decompress()` (dans le module `lzma`), 451
- `decompress()` (dans le module `zlib`), 440
- `decompress()` (méthode `bz2.BZ2Decompressor`), 447
- `decompress()` (méthode `lzma.LZMADecompressor`), 450
- `decompress()` (méthode `zlib.Decompress`), 442
- `decompressobj()` (dans le module `zlib`), 441
- décorateur, 1755
- `DEDENT` (dans le module `token`), 1651
- `dedent()` (dans le module `textwrap`), 133
- `deepcopy()` (dans le module `copy`), 242
- `def_prog_mode()` (dans le module `curses`), 652
- `def_shell_mode()` (dans le module `curses`), 652
- `default` (attribut `inspect.Parameter`), 1597
- `default` (attribut `optparse.Option`), 1732
- `default` (dans le module `email.policy`), 951

- DEFAULT (dans le module *unittest.mock*), 1440
 default() (méthode *cmd.Cmd*), 1294
 default() (méthode *json.JSONEncoder*), 991
 DEFAULT_BUFFER_SIZE (dans le module *io*), 564
 default_bufsize (dans le module *xml.dom.pulldom*), 1061
 default_exception_handler() (méthode *asyncio.AbstractEventLoop*), 866
 default_factory (attribut *collections.defaultdict*), 211
 DEFAULT_FORMAT (dans le module *tarfile*), 463
 DEFAULT_IGNORES (dans le module *filecmp*), 380
 default_open() (méthode *url-lib.request.BaseHandler*), 1111
 DEFAULT_PROTOCOL (dans le module *pickle*), 399
 default_timer() (dans le module *timeit*), 1504
 DefaultContext (classe dans *decimal*), 292
 DefaultCookiePolicy (classe dans *http.cookiejar*), 1190
 defaultdict (classe dans *collections*), 210
 DefaultDict (classe dans *typing*), 1357
 DefaultHandler() (méthode *xml.parsers.expat.xmlparser*), 1078
 DefaultHandlerExpand() (méthode *xml.parsers.expat.xmlparser*), 1078
 defaults() (méthode *configparser.ConfigParser*), 491
 DefaultSelector (classe dans *selectors*), 856
 defaultTestLoader (dans le module *unittest*), 1409
 defaultTestResult() (méthode *unittest.TestCase*), 1403
 defects (attribut *email.headerregistry.BaseHeader*), 954
 defects (attribut *email.message.EmailMessage*), 939
 defects (attribut *email.message.Message*), 974
 defpath (dans le module *os*), 561
 DefragResult (classe dans *urllib.parse*), 1126
 DefragResultBytes (classe dans *urllib.parse*), 1126
 degrees() (dans le module *math*), 276
 degrees() (dans le module *turtle*), 1271
 del
 état, 37, 73
 del_param() (méthode *email.message.EmailMessage*), 936
 del_param() (méthode *email.message.Message*), 972
 delattr() (fonction de base), 8
 delay() (dans le module *turtle*), 1283
 delay_output() (dans le module *curses*), 652
 delayload (attribut *http.cookiejar.FileCookieJar*), 1193
 delch() (méthode *curses.window*), 658
 dele() (méthode *poplib.POP3*), 1145
 delete() (méthode *ftplib.FTP*), 1142
 delete() (méthode *imaplib.IMAP4*), 1148
 delete() (méthode *tkinter.ttk.Treeview*), 1325
 DELETE_ATTR (opcode), 1670
 DELETE_DEREF (opcode), 1673
 DELETE_FAST (opcode), 1672
 DELETE_GLOBAL (opcode), 1670
 DELETE_NAME (opcode), 1670
 DELETE_SUBSCR (opcode), 1668
 deleteacl() (méthode *imaplib.IMAP4*), 1148
 deletefilehandler() (méthode *tkinter.Widget.tk*), 1314
 DeleteKey() (dans le module *winreg*), 1691
 DeleteKeyEx() (dans le module *winreg*), 1691
 deleteln() (méthode *curses.window*), 659
 deleteMe() (méthode *bdb.Breakpoint*), 1483
 DeleteValue() (dans le module *winreg*), 1692
 delimiter (attribut *csv.Dialect*), 475
 delitem() (dans le module *operator*), 344
 deliver_challenge() (dans le module *multiprocessing.connection*), 755
 delocalize() (dans le module *locale*), 1256
 demo_app() (dans le module *wsgiref.simple_server*), 1099
 denominator (attribut *fractions.Fraction*), 307
 denominator (attribut *numbers.Rational*), 270
 DeprecationWarning, 88
 deque (classe dans *collections*), 207
 Deque (classe dans *typing*), 1356
 dequeue() (méthode *logging.handlers.QueueListener*), 649
 DER_cert_to_PEM_cert() (dans le module *ssl*), 822
 derwin() (méthode *curses.window*), 659
 DES
 cipher, 1705
 descripteur, 1756
 description (attribut *sqlite3.Cursor*), 430
 description() (méthode *nntplib.NNTP*), 1156
 descriptions() (méthode *nntplib.NNTP*), 1156
 dest (attribut *optparse.Option*), 1732
 detach() (méthode *io.BufferedIOBase*), 567
 detach() (méthode *io.TextIOBase*), 570
 detach() (méthode *socket.socket*), 807
 detach() (méthode *tkinter.ttk.Treeview*), 1325
 detach() (méthode *weakref.finalize*), 234
 Detach() (méthode *winreg.PyHKEY*), 1698
 --details
 inspect command line option, 1605
 detect_api_mismatch() (dans le module *test.support*), 1480
 detect_encoding() (dans le module *tokenize*), 1654
 deterministic profiling, 1495
 device_encoding() (dans le module *os*), 526
 devnull (dans le module *os*), 561
 DEVNULL (dans le module *subprocess*), 775
 devpoll() (dans le module *select*), 847
 DevpollSelector (classe dans *selectors*), 856
 dgettext() (dans le module *gettext*), 1244
 dgettext() (dans le module *locale*), 1258

- dialect (*attribut* `csv.csvreader`), 476
- dialect (*attribut* `csv.csvwriter`), 476
- Dialect (*classe dans* `csv`), 474
- Dialog (*classe dans* `msilib`), 1688
- dict (2to3 *fixer*), 1470
- Dict (*classe dans* `typing`), 1357
- dict (*classe de base*), 73
- dict () (*méthode* `multiprocessing.managers.SyncManager`), 748
- dictConfig () (*dans le module* `logging.config`), 629
- dictionary
 - objet, 73
 - type, operations on, 73
- dictionnaire, 1756
- DictReader (*classe dans* `csv`), 473
- DictWriter (*classe dans* `csv`), 473
- diff_bytes () (*dans le module* `difflib`), 126
- diff_files (*attribut* `filecmp.dircmp`), 380
- Differ (*classe dans* `difflib`), 123, 129
- difference () (*méthode* `frozenset`), 71
- difference_update () (*méthode* `frozenset`), 72
- difflib (*module*), 122
- digest () (*méthode* `hashlib.hash`), 505
- digest () (*méthode* `hashlib.shake`), 506
- digest () (*méthode* `hmac.HMAC`), 514
- digest_size (*attribut* `hmac.HMAC`), 514
- digit () (*dans le module* `unicodedata`), 136
- digits (*dans le module* `string`), 91
- dir () (*fonction de base*), 8
- dir () (*méthode* `ftplib.FTP`), 1142
- dircmp (*classe dans* `filecmp`), 379
- directory
 - changing, 534
 - creating, 537
 - deleting, 390, 539
 - site-packages, 1605
 - traversal, 547, 549
 - walking, 547, 549
- directory ...
 - compileall command line option, 1660
- Directory (*classe dans* `msilib`), 1687
- DirEntry (*classe dans* `os`), 541
- DirList (*classe dans* `tkinter.tix`), 1332
- dirname () (*dans le module* `os.path`), 368
- DirSelectBox (*classe dans* `tkinter.tix`), 1332
- DirSelectDialog (*classe dans* `tkinter.tix`), 1332
- DirTree (*classe dans* `tkinter.tix`), 1332
- dis (*module*), 1662
- dis () (*dans le module* `dis`), 1664
- dis () (*dans le module* `pickletools`), 1676
- dis () (*méthode* `dis.Bytecode`), 1663
- disable (`pdb` *command*), 1492
- disable () (*dans le module* `faulthandler`), 1488
- disable () (*dans le module* `gc`), 1588
- disable () (*dans le module* `logging`), 626
- disable () (*méthode* `bdb.Breakpoint`), 1484
- disable () (*méthode* `profile.Profile`), 1498
- disable_interspersed_args () (*méthode* `optparse.OptionParser`), 1736
- DisableReflectionKey () (*dans le module* `winreg`), 1694
- disassemble () (*dans le module* `dis`), 1664
- discard (*attribut* `http.cookiejar.Cookie`), 1196
- discard () (*méthode* `frozenset`), 72
- discard () (*méthode* `mailbox.Mailbox`), 996
- discard () (*méthode* `mailbox.MH`), 1001
- discard_buffers () (*méthode* `asyncio.async_chat`), 919
- disco () (*dans le module* `dis`), 1664
- discover () (*méthode* `unittest.TestLoader`), 1406
- disk_usage () (*dans le module* `shutil`), 391
- dispatch_call () (*méthode* `bdb.Bdb`), 1484
- dispatch_exception () (*méthode* `bdb.Bdb`), 1485
- dispatch_line () (*méthode* `bdb.Bdb`), 1484
- dispatch_return () (*méthode* `bdb.Bdb`), 1485
- dispatch_table (*attribut* `pickle.Pickler`), 400
- dispatcher (*classe dans* `asyncore`), 915
- dispatcher_with_send (*classe dans* `asyncore`), 917
- display (`pdb` *command*), 1494
- display_name (*attribut* `email.headerregistry.Address`), 957
- display_name (*attribut* `email.headerregistry.Group`), 957
- displayhook () (*dans le module* `sys`), 1540
- dist () (*dans le module* `platform`), 675
- distance () (*dans le module* `turtle`), 1271
- distb () (*dans le module* `dis`), 1664
- distribution simple, 1763
- distutils (*module*), 1521
- divide () (*méthode* `decimal.Context`), 294
- divide_int () (*méthode* `decimal.Context`), 295
- division entière, 1757
- DivisionByZero (*classe dans* `decimal`), 298
- divmod () (*fonction de base*), 9
- divmod () (*méthode* `decimal.Context`), 295
- DllCanUnloadNow () (*dans le module* `ctypes`), 707
- DllGetClassObject () (*dans le module* `ctypes`), 707
- dllhandle (*dans le module* `sys`), 1540
- dngettext () (*dans le module* `gettext`), 1244
- do_clear () (*méthode* `bdb.Bdb`), 1485
- do_command () (*méthode* `curses.textpad.Textbox`), 668
- do_GET () (*méthode* `http.server.SimpleHTTPRequestHandler`), 1185
- do_handshake () (*méthode* `ssl.SSLSocket`), 829
- do_HEAD () (*méthode* `http.server.SimpleHTTPRequestHandler`), 1184
- do_POST () (*méthode* `http.server.CGIHTTPRequestHandler`), 1186

- `doc` (attribut `json.JSONDecodeError`), 991
`doc_header` (attribut `cmd.Cmd`), 1295
`DocCGIXMLRPCRequestHandler` (classe dans `xmlrpc.server`), 1210
`DocFileSuite()` (dans le module `doctest`), 1376
`doCleanups()` (méthode `unittest.TestCase`), 1403
`doccmd()` (méthode `smtplib.SMTP`), 1161
`docstring` (attribut `doctest.DocTest`), 1379
`docstring` (chaîne de documentation), 1756
`DocTest` (classe dans `doctest`), 1378
`doctest` (module), 1363
`DocTestFailure`, 1384
`DocTestFinder` (classe dans `doctest`), 1380
`DocTestParser` (classe dans `doctest`), 1380
`DocTestRunner` (classe dans `doctest`), 1381
`DocTestSuite()` (dans le module `doctest`), 1377
`doctype()` (méthode `xml.etree.ElementTree.TreeBuilder`), 1043
`doctype()` (méthode `xml.etree.ElementTree.XMLParser`), 1043
`documentation`
 generation, 1362
 online, 1362
`documentElement` (attribut `xml.dom.Document`), 1050
`DocXMLRPCRequestHandler` (classe dans `xmlrpc.server`), 1210
`DocXMLRPCServer` (classe dans `xmlrpc.server`), 1210
`domain` (attribut `email.headerregistry.Address`), 957
`domain` (attribut `tracemalloc.DomainFilter`), 1516
`domain` (attribut `tracemalloc.Filter`), 1516
`domain_initial_dot` (attribut `http.cookiejar.Cookie`), 1197
`domain_return_ok()` (méthode `http.cookiejar.CookiePolicy`), 1193
`domain_specified` (attribut `http.cookiejar.Cookie`), 1197
`DomainFilter` (classe dans `tracemalloc`), 1516
`DomainLiberal` (attribut `http.cookiejar.DefaultCookiePolicy`), 1196
`DomainRFC2965Match` (attribut `http.cookiejar.DefaultCookiePolicy`), 1196
`DomainStrict` (attribut `http.cookiejar.DefaultCookiePolicy`), 1196
`DomainStrictNoDots` (attribut `http.cookiejar.DefaultCookiePolicy`), 1195
`DomainStrictNonDomain` (attribut `http.cookiejar.DefaultCookiePolicy`), 1195
`DOMEventStream` (classe dans `xml.dom.pulldom`), 1061
`DOMException`, 1053
`DomStringSizeErr`, 1053
`done()` (dans le module `turtle`), 1284
`done()` (méthode `asyncio.Future`), 876
`done()` (méthode `concurrent.futures.Future`), 772
`done()` (méthode `xdrlib.Unpacker`), 497
`DONT_ACCEPT_BLANKLINE` (dans le module `doctest`), 1371
`DONT_ACCEPT_TRUE_FOR_1` (dans le module `doctest`), 1370
`dont_write_bytecode` (dans le module `sys`), 1541
`doRollover()` (méthode `logging.handlers.RotatingFileHandler`), 641
`doRollover()` (méthode `logging.handlers.TimedRotatingFileHandler`), 642
`DOT` (dans le module `token`), 1651
`dot()` (dans le module `turtle`), 1268
`DOTALL` (dans le module `re`), 109
`doublequote` (attribut `csv.Dialect`), 475
`DOUBLESASH` (dans le module `token`), 1651
`DOUBLESASHEQUAL` (dans le module `token`), 1651
`DOUBLESTAR` (dans le module `token`), 1651
`DOUBLESTAREQUAL` (dans le module `token`), 1651
`doupdate()` (dans le module `curses`), 652
`down` (`pdb` command), 1492
`down()` (dans le module `turtle`), 1272
`drain()` (méthode `asyncio.StreamWriter`), 894
`drop_whitespace` (attribut `textwrap.TextWrapper`), 134
`dropwhile()` (dans le module `itertools`), 326
`dst()` (méthode `datetime.datetime`), 181
`dst()` (méthode `datetime.time`), 187
`dst()` (méthode `datetime.timezone`), 195
`dst()` (méthode `datetime.tzinfo`), 188
`DTDHandler` (classe dans `xml.sax.handler`), 1064
`duck-typing`, 1756
`DumbWriter` (classe dans `formatter`), 1681
`dummy_threading` (module), 793
`dump()` (dans le module `ast`), 1648
`dump()` (dans le module `json`), 987
`dump()` (dans le module `marshal`), 414
`dump()` (dans le module `pickle`), 399
`dump()` (dans le module `plistlib`), 499
`dump()` (dans le module `xml.etree.ElementTree`), 1037
`dump()` (méthode `pickle.Pickler`), 400
`dump()` (méthode `tracemalloc.Snapshot`), 1517
`dump_stats()` (méthode `profile.Profile`), 1498
`dump_stats()` (méthode `pstats.Stats`), 1499
`dump_traceback()` (dans le module `faulthandler`), 1487
`dump_traceback_later()` (dans le module `faulthandler`), 1488
`dumps()` (dans le module `json`), 988
`dumps()` (dans le module `marshal`), 414
`dumps()` (dans le module `pickle`), 399
`dumps()` (dans le module `plistlib`), 500
`dumps()` (dans le module `xmlrpc.client`), 1204
`dup()` (dans le module `os`), 526
`dup()` (méthode `socket.socket`), 807

`dup2()` (dans le module `os`), 526
`DUP_TOP` (opcode), 1666
`DUP_TOP_TWO` (opcode), 1666
`DuplicateOptionError`, 494
`DuplicateSectionError`, 494
`dwFlags` (attribut `subprocess.STARTUPINFO`), 782
`DynamicClassAttribute()` (dans le module `types`), 242

E

`-e`
 `tokenize` command line option, 1655
`e` (dans le module `cmath`), 280
`e` (dans le module `math`), 277
`-e <tarfile> [<output_dir>]`
 `tarfile` command line option, 468
`-e <zipfile> <output_dir>`
 `zipfile` command line option, 460
`E2BIG` (dans le module `errno`), 676
`EACCES` (dans le module `errno`), 676
`EADDRINUSE` (dans le module `errno`), 680
`EADDRNOTAVAIL` (dans le module `errno`), 680
`EADV` (dans le module `errno`), 679
`EAFNOSUPPORT` (dans le module `errno`), 680
`EAFP`, 1756
`EAGAIN` (dans le module `errno`), 676
`EALREADY` (dans le module `errno`), 681
`east_asian_width()` (dans le module `unicodedata`), 136
`EBADE` (dans le module `errno`), 678
`EBADF` (dans le module `errno`), 676
`EBADFD` (dans le module `errno`), 679
`EBADMSG` (dans le module `errno`), 679
`EBADR` (dans le module `errno`), 678
`EBADRQC` (dans le module `errno`), 678
`EBADSLT` (dans le module `errno`), 678
`EBFONT` (dans le module `errno`), 678
`EBUSY` (dans le module `errno`), 676
`ECHILD` (dans le module `errno`), 676
`echo()` (dans le module `curses`), 652
`echochar()` (méthode `curses.window`), 659
`ECHRNQ` (dans le module `errno`), 677
`ECOMM` (dans le module `errno`), 679
`ECONNABORTED` (dans le module `errno`), 680
`ECONNREFUSED` (dans le module `errno`), 681
`ECONNRESET` (dans le module `errno`), 680
`EDEADLK` (dans le module `errno`), 677
`EDEADLOCK` (dans le module `errno`), 678
`EDESTADDRREQ` (dans le module `errno`), 680
`edit()` (méthode `curses.textpad.Textbox`), 668
`EDOM` (dans le module `errno`), 677
`EDOTDOT` (dans le module `errno`), 679
`EDQUOT` (dans le module `errno`), 681
`EEXIST` (dans le module `errno`), 676

`EFAULT` (dans le module `errno`), 676
`EFBIG` (dans le module `errno`), 677
`effective()` (dans le module `bdb`), 1487
`ehlo()` (méthode `smtplib.SMTP`), 1161
`ehlo_or_helo_if_needed()` (méthode `smtplib.SMTP`), 1161
`EHOSTDOWN` (dans le module `errno`), 681
`EHOSTUNREACH` (dans le module `errno`), 681
`EIDRM` (dans le module `errno`), 677
`EILSEQ` (dans le module `errno`), 679
`EINPROGRESS` (dans le module `errno`), 681
`EINTR` (dans le module `errno`), 676
`EINVAL` (dans le module `errno`), 676
`EIO` (dans le module `errno`), 676
`EISCONN` (dans le module `errno`), 680
`EISDIR` (dans le module `errno`), 676
`EISNAM` (dans le module `errno`), 681
`EL2HLT` (dans le module `errno`), 678
`EL2NSYNC` (dans le module `errno`), 677
`EL3HLT` (dans le module `errno`), 678
`EL3RST` (dans le module `errno`), 678
`Element` (classe dans `xml.etree.ElementTree`), 1039
`element_create()` (méthode `tkinter.ttk.Style`), 1329
`element_names()` (méthode `tkinter.ttk.Style`), 1329
`element_options()` (méthode `tkinter.ttk.Style`), 1329
`ElementDeclHandler()` (méthode `xml.parsers.expat.xmlparser`), 1077
`elements()` (méthode `collections.Counter`), 205
`ElementTree` (classe dans `xml.etree.ElementTree`), 1041
`ELIBACC` (dans le module `errno`), 679
`ELIBBAD` (dans le module `errno`), 679
`ELIBEXEC` (dans le module `errno`), 679
`ELIBMAX` (dans le module `errno`), 679
`ELIBSCN` (dans le module `errno`), 679
`Ellinghouse, Lance`, 1021
`ELLIPSIS` (dans le module `doctest`), 1371
`ELLIPSIS` (dans le module `token`), 1651
`Ellipsis` (variable de base), 25
`ELNRNG` (dans le module `errno`), 678
`ELOOP` (dans le module `errno`), 677
`email` (module), 931
 `email.charset` (module), 979
 `email.contentmanager` (module), 958
 `email.encoders` (module), 981
 `email.errors` (module), 952
 `email.generator` (module), 943
 `email.header` (module), 977
 `email.headerregistry` (module), 953
 `email.iterators` (module), 984
 `EmailMessage` (classe dans `email.message`), 933
 `email.message` (module), 932
 `email.mime` (module), 974
 `email.parser` (module), 940
 `EmailPolicy` (classe dans `email.policy`), 949

- email.policy (module), 946
email.utils (module), 982
EMFILE (dans le module errno), 677
emit() (méthode logging.FileHandler), 639
emit() (méthode logging.Handler), 619
emit() (méthode logging.handlers.BufferingHandler), 647
emit() (méthode logging.handlers.DatagramHandler), 644
emit() (méthode logging.handlers.HTTPHandler), 648
emit() (méthode logging.handlers.NTEventLogHandler), 646
emit() (méthode logging.handlers.QueueHandler), 648
emit() (méthode logging.handlers.RotatingFileHandler), 641
emit() (méthode logging.handlers.SMTPHandler), 646
emit() (méthode logging.handlers.SocketHandler), 643
emit() (méthode logging.handlers.SysLogHandler), 644
emit() (méthode logging.handlers.TimedRotatingFileHandler), 642
emit() (méthode logging.handlers.WatchedFileHandler), 640
emit() (méthode logging.NullHandler), 639
emit() (méthode logging.StreamHandler), 639
EMLINK (dans le module errno), 677
Empty, 791
empty (attribut inspect.Parameter), 1597
empty (attribut inspect.Signature), 1596
empty() (méthode asyncio.Queue), 907
empty() (méthode multiprocessing.Queue), 737
empty() (méthode multiprocessing.SimpleQueue), 738
empty() (méthode queue.Queue), 791
empty() (méthode sched.scheduler), 790
EMPTY_NAMESPACE (dans le module xml.dom), 1046
emptyline() (méthode cmd.Cmd), 1294
EMSGSIZE (dans le module errno), 680
EMULTIHOP (dans le module errno), 679
enable (pdb command), 1492
enable() (dans le module cgiitb), 1095
enable() (dans le module faulthandler), 1488
enable() (dans le module gc), 1588
enable() (méthode bdb.Breakpoint), 1483
enable() (méthode imaplib.IMAP4), 1148
enable() (méthode profile.Profile), 1498
enable_callback_tracebacks() (dans le module sqlite3), 422
enable_interspersed_args() (méthode optparse.OptionParser), 1736
enable_load_extension() (méthode sqlite3.Connection), 425
enable_traversal() (méthode tkinter.ttk.Notebook), 1321
ENABLE_USER_SITE (dans le module site), 1607
EnableReflectionKey() (dans le module winreg), 1695
ENAMETOOLONG (dans le module errno), 677
ENAVAIL (dans le module errno), 681
enclose() (méthode curses.window), 659
encodage de texte, 1763
encode
 Codecs, 151
encode (attribut codecs.CodecInfo), 151
encode() (dans le module base64), 1017
encode() (dans le module codecs), 151
encode() (dans le module quopri), 1021
encode() (dans le module uu), 1021
encode() (méthode codecs.Codec), 155
encode() (méthode codecs.IncrementalEncoder), 156
encode() (méthode email.header.Header), 978
encode() (méthode json.JSONEncoder), 991
encode() (méthode str), 42
encode() (méthode xmlrpc.client.Binary), 1201
encode() (méthode xmlrpc.client.DateTime), 1201
encode_7or8bit() (dans le module email.encoders), 981
encode_base64() (dans le module email.encoders), 981
encode_noop() (dans le module email.encoders), 981
encode_quopri() (dans le module email.encoders), 981
encode_rfc2231() (dans le module email.utils), 984
encodebytes() (dans le module base64), 1017
EncodedFile() (dans le module codecs), 153
encodePriority() (méthode logging.handlers.SysLogHandler), 645
encodestring() (dans le module base64), 1017
encodestring() (dans le module quopri), 1021
encoding
 base64, 1015
 quoted-printable, 1021
encoding (attribut curses.window), 659
encoding (attribut io.TextIOBase), 570
encoding (attribut UnicodeError), 86
ENCODING (dans le module tarfile), 462
ENCODING (dans le module tokenize), 1654
encodings_map (attribut mimetypes.MimeTypes), 1014
encodings_map (dans le module mimetypes), 1013
encodings.idna (module), 166
encodings.mbcx (module), 166
encodings.utf_8_sig (module), 167
end (attribut UnicodeError), 86
end() (méthode re.match), 116
end() (méthode xml.etree.ElementTree.TreeBuilder), 1043
end_fill() (dans le module turtle), 1275
END_FINALLY (opcode), 1669

`end_headers()` (méthode `http.server.BaseHTTPRequestHandler`), 1184
`end_paragraph()` (méthode `formatter.formatter`), 1678
`end_poly()` (dans le module `turtle`), 1280
`EndCdataSectionHandler()` (méthode `xml.parsers.expat.xmlparser`), 1078
`EndDoctypeDeclHandler()` (méthode `xml.parsers.expat.xmlparser`), 1077
`endDocument()` (méthode `xml.sax.handler.ContentHandler`), 1066
`endElement()` (méthode `xml.sax.handler.ContentHandler`), 1067
`EndElementHandler()` (méthode `xml.parsers.expat.xmlparser`), 1077
`EndElementNS()` (méthode `xml.sax.handler.ContentHandler`), 1067
`endheaders()` (méthode `http.client.HTTPConnection`), 1136
`ENDMARKER` (dans le module `token`), 1651
`EndNamespaceDeclHandler()` (méthode `xml.parsers.expat.xmlparser`), 1078
`endpos` (attribut `re.match`), 116
`endPrefixMapping()` (méthode `xml.sax.handler.ContentHandler`), 1066
`endswith()` (méthode `bytearray`), 54
`endswith()` (méthode `bytes`), 54
`endswith()` (méthode `str`), 43
`endwin()` (dans le module `curses`), 652
`ENETDOWN` (dans le module `errno`), 680
`ENETRESET` (dans le module `errno`), 680
`ENETUNREACH` (dans le module `errno`), 680
`ENFILE` (dans le module `errno`), 676
`ENOANO` (dans le module `errno`), 678
`ENOBUFFS` (dans le module `errno`), 680
`ENOCSS` (dans le module `errno`), 678
`ENODATA` (dans le module `errno`), 678
`ENODEV` (dans le module `errno`), 676
`ENOENT` (dans le module `errno`), 675
`ENOEXEC` (dans le module `errno`), 676
`ENOLCK` (dans le module `errno`), 677
`ENOLINK` (dans le module `errno`), 679
`ENOMEM` (dans le module `errno`), 676
`ENOMSG` (dans le module `errno`), 677
`ENONET` (dans le module `errno`), 678
`ENOPKG` (dans le module `errno`), 678
`ENOPROTOOPT` (dans le module `errno`), 680
`ENOSPC` (dans le module `errno`), 677
`ENOSR` (dans le module `errno`), 678
`ENOSTR` (dans le module `errno`), 678
`ENOSYS` (dans le module `errno`), 677
`ENOTBLK` (dans le module `errno`), 676
`ENOTCONN` (dans le module `errno`), 680
`ENOTDIR` (dans le module `errno`), 676
`ENOTEMPTY` (dans le module `errno`), 677
`ENOTNAM` (dans le module `errno`), 681
`ENOTSOCK` (dans le module `errno`), 680
`ENOTTY` (dans le module `errno`), 677
`ENOTUNIQ` (dans le module `errno`), 679
`enqueue()` (méthode `logging.handlers.QueueHandler`), 648
`enqueue_sentinel()` (méthode `logging.handlers.QueueListener`), 649
`ensure_directories()` (méthode `venv.EnvBuilder`), 1527
`ensure_future()` (dans le module `asyncio`), 880
`ensurepip` (module), 1522
`enter()` (méthode `sched.scheduler`), 790
`enter_context()` (méthode `contextlib.ExitStack`), 1567
`enterabs()` (méthode `sched.scheduler`), 790
`entities` (attribut `xml.dom.DocumentType`), 1050
`EntityDeclHandler()` (méthode `xml.parsers.expat.xmlparser`), 1077
`entitydefs` (dans le module `html.entities`), 1028
`EntityResolver` (classe dans `xml.sax.handler`), 1064
`entrée de chemin`, 1761
`Enum` (classe dans `enum`), 251
`enum` (module), 251
`enum_certificates()` (dans le module `ssl`), 823
`enum_crls()` (dans le module `ssl`), 823
`enumerate()` (dans le module `threading`), 716
`enumerate()` (fonction de base), 9
`EnumKey()` (dans le module `winreg`), 1692
`EnumValue()` (dans le module `winreg`), 1692
`EnvBuilder` (classe dans `venv`), 1526
`environ` (dans le module `os`), 520
`environ` (dans le module `posix`), 1702
`environb` (dans le module `os`), 521
`environment variables`
 deleting, 525
 setting, 523
`EnvironmentError`, 87
`Environments`
 virtual, 1523
`EnvironmentVarGuard` (classe dans `test.support`), 1480
`environnement virtuel`, 1764
`ENXIO` (dans le module `errno`), 676
`eof` (attribut `bz2.BZ2Decompressor`), 447
`eof` (attribut `lzma.LZMADecompressor`), 451
`eof` (attribut `shlex.shlex`), 1301
`eof` (attribut `ssl.MemoryBIO`), 844
`eof` (attribut `zlib.Decompress`), 442
`eof_received()` (méthode `asyncio.Protocol`), 887
`EOFError`, 83
`EOPNOTSUPP` (dans le module `errno`), 680
`EOVERFLOW` (dans le module `errno`), 679

- EPERM (dans le module *errno*), 675
 EPFNOSUPPORT (dans le module *errno*), 680
 epilogue (attribut *email.message.EmailMessage*), 939
 epilogue (attribut *email.message.Message*), 974
 EPIPE (dans le module *errno*), 677
 epoch, 573
 epoll() (dans le module *select*), 847
 EpollSelector (classe dans *selectors*), 856
 EPROTO (dans le module *errno*), 679
 EPROTONOSUPPORT (dans le module *errno*), 680
 EPROTOTYPE (dans le module *errno*), 680
 eq() (dans le module *operator*), 342
 EQEQUAL (dans le module *token*), 1651
 EQUAL (dans le module *token*), 1651
 ERA (dans le module locale), 1255
 ERA_D_FMT (dans le module locale), 1255
 ERA_D_T_FMT (dans le module locale), 1255
 ERA_T_FMT (dans le module locale), 1255
 ERANGE (dans le module *errno*), 677
 erase() (méthode *curses.window*), 659
 erasechar() (dans le module *curses*), 652
 EREMCHG (dans le module *errno*), 679
 EREMOTE (dans le module *errno*), 678
 EREMOTEIO (dans le module *errno*), 681
 ERESTART (dans le module *errno*), 679
 erf() (dans le module *math*), 276
 erfc() (dans le module *math*), 276
 EROFS (dans le module *errno*), 677
 ERR (dans le module *curses*), 663
 errcheck (attribut *ctypes.FuncPtr*), 704
 errcode (attribut *xmlrpc.client.ProtocolError*), 1203
 errmsg (attribut *xmlrpc.client.ProtocolError*), 1203
 errno
 module, 84
 errno (attribut *OSError*), 84
 errno (module), 675
 Error, 391, 431, 475, 494, 498, 1011, 1018, 1020, 1022, 1086, 1229, 1232, 1251
 error, 112, 146, 242, 415418, 439, 519, 613, 651, 794, 799, 847, 1074, 1223, 1712, 1716
 error() (dans le module *logging*), 625
 error() (méthode *argparse.ArgumentParser*), 611
 error() (méthode *logging.Logger*), 617
 error() (méthode *urllib.request.OpenerDirector*), 1111
 error() (méthode *xml.sax.handler.ErrorHandler*), 1068
 error_body (attribut *wsgiref.handlers.BaseHandler*), 1103
 error_content_type (attribut *http.server.BaseHTTPRequestHandler*), 1182
 error_headers (attribut *wsgiref.handlers.BaseHandler*), 1103
 error_leader() (méthode *shlex.shlex*), 1300
 error_message_format (attribut *http.server.BaseHTTPRequestHandler*), 1182
 error_output() (méthode *wsgiref.handlers.BaseHandler*), 1103
 error_perm, 1140
 error_proto, 1140, 1144
 error_received() (méthode *asyncio.DatagramProtocol*), 887
 error_reply, 1140
 error_status (attribut *wsgiref.handlers.BaseHandler*), 1103
 error_temp, 1140
 ErrorByteIndex (attribut *xml.parsers.expat.xmlparser*), 1076
 ErrorCode (attribut *xml.parsers.expat.xmlparser*), 1076
 errorcode (dans le module *errno*), 675
 ErrorColumnNumber (attribut *xml.parsers.expat.xmlparser*), 1076
 ErrorHandler (classe dans *xml.sax.handler*), 1064
 ErrorLineNumber (attribut *xml.parsers.expat.xmlparser*), 1076
 Errors
 logging, 614
 errors (attribut *io.TextIOBase*), 570
 errors (attribut *unittest.TestLoader*), 1405
 errors (attribut *unittest.TestResult*), 1407
 ErrorString() (dans le module *xml.parsers.expat*), 1074
 ERRORTOKEN (dans le module *token*), 1651
 escape (attribut *shlex.shlex*), 1300
 escape() (dans le module *cgi*), 1092
 escape() (dans le module *glob*), 385
 escape() (dans le module *html*), 1023
 escape() (dans le module *re*), 112
 escape() (dans le module *xml.sax.saxutils*), 1069
 escapechar (attribut *csv.Dialect*), 475
 escapedquotes (attribut *shlex.shlex*), 1300
 ESHUTDOWN (dans le module *errno*), 680
 ESOCKTNOSUPPORT (dans le module *errno*), 680
 espace de nommage, 1760
 ESPIPE (dans le module *errno*), 677
 ESRCH (dans le module *errno*), 675
 ESRMNT (dans le module *errno*), 679
 ESTALE (dans le module *errno*), 681
 ESTRPIPE (dans le module *errno*), 679
 état
 assert, 83
 del, 37, 73
 except, 81
 if, 27
 import, 23, 1606, 1745
 raise, 81
 try, 81
 while, 27
 ETIME (dans le module *errno*), 678
 ETIMEDOUT (dans le module *errno*), 681

- Etiny() (méthode decimal.Context), 294
- ETOOMANYREFS (dans le module errno), 681
- Etop() (méthode decimal.Context), 294
- ETXTBSY (dans le module errno), 677
- EUCLEAN (dans le module errno), 681
- EUNATCH (dans le module errno), 678
- EUSERS (dans le module errno), 679
- eval
 - fonction de base, 78, 245, 1641
- eval() (fonction de base), 10
- Event (classe dans asyncio), 905
- Event (classe dans multiprocessing), 742
- Event (classe dans threading), 724
- event scheduling, 789
- event() (méthode msilib.Control), 1687
- Event() (méthode multiprocessing.sing.managers.SyncManager), 748
- events (attribut selectors.SelectorKey), 854
- events (widgets), 1312
- EWouldBlock (dans le module errno), 677
- EX_CANTCREAT (dans le module os), 552
- EX_CONFIG (dans le module os), 553
- EX_DATAERR (dans le module os), 552
- EX_IOERR (dans le module os), 552
- EX_NOHOST (dans le module os), 552
- EX_NOINPUT (dans le module os), 552
- EX_NOPERM (dans le module os), 552
- EX_NOTFOUND (dans le module os), 553
- EX_NOUSER (dans le module os), 552
- EX_OK (dans le module os), 551
- EX_OSERR (dans le module os), 552
- EX_OSFILE (dans le module os), 552
- EX_PROTOCOL (dans le module os), 552
- EX_SOFTWARE (dans le module os), 552
- EX_TEMPFAIL (dans le module os), 552
- EX_UNAVAILABLE (dans le module os), 552
- EX_USAGE (dans le module os), 552
- exact
 - tokenize command line option, 1655
- example (attribut doctest.DocTestFailure), 1384
- example (attribut doctest.UnexpectedException), 1385
- Example (classe dans doctest), 1379
- examples (attribut doctest.DocTest), 1378
- exc_info (attribut doctest.UnexpectedException), 1385
- exc_info() (dans le module sys), 1541
- exc_msg (attribut doctest.Example), 1379
- exc_type (attribut traceback.TracebackException), 1583
- excel (classe dans csv), 474
- excel_tab (classe dans csv), 474
- except
 - état, 81
- except (2to3 fixer), 1470
- excepthook() (dans le module sys), 1541
- excepthook() (in module sys), 1095
- Exception, 82
- EXCEPTION (dans le module tkinter), 1314
- exception() (dans le module logging), 626
- exception() (méthode asyncio.Future), 876
- exception() (méthode asyncio.StreamReader), 893
- exception() (méthode concurrent.futures.Future), 772
- exception() (méthode logging.Logger), 617
- exceptions
 - in CGI scripts, 1095
- EXDEV (dans le module errno), 676
- exec
 - fonction de base, 10, 78, 1641
- exec (2to3 fixer), 1470
- exec() (fonction de base), 10
- exec_module() (méthode importlib.abc.InspectLoader), 1628
- exec_module() (méthode importlib.abc.Loader), 1626
- exec_module() (méthode importlib.abc.SourceLoader), 1629
- exec_module() (méthode importlib.machinery.ExtensionFileLoader), 1633
- exec_prefix (dans le module sys), 1541
- execfile (2to3 fixer), 1470
- execl() (dans le module os), 550
- execle() (dans le module os), 550
- execlp() (dans le module os), 550
- execlpe() (dans le module os), 550
- executable (dans le module sys), 1542
- Executable Zip Files, 1531
- Execute() (méthode msilib.View), 1685
- execute() (méthode sqlite3.Connection), 423
- execute() (méthode sqlite3.Cursor), 427
- executemany() (méthode sqlite3.Connection), 423
- executemany() (méthode sqlite3.Cursor), 428
- executescript() (méthode sqlite3.Connection), 423
- executescript() (méthode sqlite3.Cursor), 428
- ExecutionLoader (classe dans importlib.abc), 1628
- Executor (classe dans concurrent.futures), 768
- execv() (dans le module os), 550
- execve() (dans le module os), 550
- execvp() (dans le module os), 550
- execvpe() (dans le module os), 550
- ExFileSelectBox (classe dans tkinter.tix), 1333
- EXFULL (dans le module errno), 678
- exists() (dans le module os.path), 368
- exists() (méthode pathlib.Path), 362
- exists() (méthode tkinter.ttk.Treeview), 1325
- exit (variable de base), 26
- exit() (dans le module _thread), 794
- exit() (dans le module sys), 1542
- exit() (méthode argparse.ArgumentParser), 611
- exitcode (attribut multiprocessing.Process), 734
- exitfunc (2to3 fixer), 1470
- exitonclick() (dans le module turtle), 1287

- ExitStack (classe dans *contextlib*), 1567
- exp () (dans le module *cmath*), 279
- exp () (dans le module *math*), 274
- exp () (méthode *decimal.Context*), 295
- exp () (méthode *decimal.Decimal*), 288
- expand () (méthode *re.match*), 114
- expand_tabs (attribut *textwrap.TextWrapper*), 134
- ExpandEnvironmentStrings () (dans le module *winreg*), 1692
- expandNode () (méthode *xml.dom.pulldom.DOMEventStream*), 1061
- expandtabs () (méthode *bytearray*), 58
- expandtabs () (méthode *bytes*), 58
- expandtabs () (méthode *str*), 43
- expanduser () (dans le module *os.path*), 368
- expanduser () (méthode *pathlib.Path*), 362
- expandvars () (dans le module *os.path*), 369
- Expat, 1074
- ExpatriError, 1074
- expect () (méthode *telnetlib.Telnet*), 1169
- expected (attribut *asyncio.IncompleteReadError*), 895
- expectedFailure () (dans le module *unittest*), 1394
- expectedFailures (attribut *unittest.TestResult*), 1407
- expires (attribut *http.cookiejar.Cookie*), 1196
- exploded (attribut *ipaddress.IPv4Address*), 1212
- exploded (attribut *ipaddress.IPv4Network*), 1216
- exploded (attribut *ipaddress.IPv6Address*), 1213
- exploded (attribut *ipaddress.IPv6Network*), 1218
- expm1 () (dans le module *math*), 274
- expovariate () (dans le module *random*), 311
- expr () (dans le module *parser*), 1640
- expression, 1756
- expression génératrice, 1757
- expunge () (méthode *imaplib.IMAP4*), 1148
- extend () (méthode *array.array*), 230
- extend () (méthode *collections.deque*), 208
- extend () (méthode *xml.etree.ElementTree.Element*), 1040
- extend () (sequence method), 37
- extend_path () (dans le module *pkgutil*), 1617
- EXTENDED_ARG (opcode), 1673
- ExtendedContext (classe dans *decimal*), 292
- ExtendedInterpolation (classe dans *configparser*), 482
- extendleft () (méthode *collections.deque*), 208
- EXTENSION_SUFFIXES (dans le module *importlib.machinery*), 1630
- ExtensionFileLoader (classe dans *importlib.machinery*), 1632
- extensions_map (attribut *http.server.SimpleHTTPRequestHandler*), 1184
- External Data Representation, 398, 496
- external_attr (attribut *zipfile.ZipInfo*), 459
- ExternalClashError, 1011
- ExternalEntityParserCreate () (méthode *xml.parsers.expat.xmlparser*), 1075
- ExternalEntityRefHandler () (méthode *xml.parsers.expat.xmlparser*), 1078
- extra (attribut *zipfile.ZipInfo*), 459
- extract <tarfile> [<output_dir>] tarfile command line option, 468
- extract () (méthode de la classe *traceback.StackSummary*), 1584
- extract () (méthode *tarfile.TarFile*), 464
- extract () (méthode *zipfile.ZipFile*), 456
- extract_cookies () (méthode *http.cookiejar.CookieJar*), 1191
- extract_stack () (dans le module *traceback*), 1582
- extract_tb () (dans le module *traceback*), 1581
- extract_version (attribut *zipfile.ZipInfo*), 459
- extractall () (méthode *tarfile.TarFile*), 464
- extractall () (méthode *zipfile.ZipFile*), 456
- ExtractError, 462
- extractfile () (méthode *tarfile.TarFile*), 465
- extsep (dans le module *os*), 561
- ## F
- f compileall command line option, 1660
trace command line option, 1509
unittest command line option, 1389
- f-string, 1756
- f_contiguous (attribut *memoryview*), 70
- F_LOCK (dans le module *os*), 528
- F_OK (dans le module *os*), 534
- F_TEST (dans le module *os*), 528
- F_TLOCK (dans le module *os*), 528
- F_ULOCK (dans le module *os*), 528
- fabs () (dans le module *math*), 273
- factorial () (dans le module *math*), 273
- factory () (méthode de la classe *importlib.util.LazyLoader*), 1636
- fail () (méthode *unittest.TestCase*), 1402
- FAIL_FAST (dans le module *doctest*), 1372
- failfast unittest command line option, 1389
- failfast (attribut *unittest.TestResult*), 1408
- failureException (attribut *unittest.TestCase*), 1402
- failures (attribut *unittest.TestResult*), 1407
- FakePath (classe dans *test.support*), 1481
- False, 27, 79
- false, 27
- False (Built-in object), 27
- False (variable de base), 25
- family (attribut *socket.socket*), 813
- FancyURLopener (classe dans *urllib.request*), 1120
- fast (attribut *pickle.Pickler*), 401

`fatalError()` (méthode `xml.sax.handler.ErrorHandler`), 1068
`Fault` (classe dans `xmlrpc.client`), 1202
`faultCode` (attribut `xmlrpc.client.Fault`), 1202
`faulthandler` (module), 1487
`faultString` (attribut `xmlrpc.client.Fault`), 1202
`fchdir()` (dans le module `os`), 536
`fchmod()` (dans le module `os`), 526
`fchown()` (dans le module `os`), 527
`FCICreate()` (dans le module `msilib`), 1683
`fcntl` (module), 1709
`fcntl()` (dans le module `fcntl`), 1709
`fd` (attribut `selectors.SelectorKey`), 854
`fd()` (dans le module `turtle`), 1266
`fdatasync()` (dans le module `os`), 527
`fdopen()` (dans le module `os`), 526
`Feature` (classe dans `msilib`), 1687
`feature_external_ges` (dans le module `xml.sax.handler`), 1065
`feature_external_pes` (dans le module `xml.sax.handler`), 1065
`feature_namespace_prefixes` (dans le module `xml.sax.handler`), 1064
`feature_namespaces` (dans le module `xml.sax.handler`), 1064
`feature_string_interning` (dans le module `xml.sax.handler`), 1064
`feature_validation` (dans le module `xml.sax.handler`), 1065
`feed()` (méthode `email.parser.BytesFeedParser`), 941
`feed()` (méthode `html.parser.HTMLParser`), 1025
`feed()` (méthode `xml.etree.ElementTree.XMLParser`), 1043
`feed()` (méthode `xml.etree.ElementTree.XMLPullParser`), 1044
`feed()` (méthode `xml.sax.xmlreader.IncrementalParser`), 1072
`feed_data()` (méthode `asyncio.StreamReader`), 893
`feed_eof()` (méthode `asyncio.StreamReader`), 893
`FeedParser` (classe dans `email.parser`), 941
`fetch()` (méthode `imaplib.IMAP4`), 1149
`Fetch()` (méthode `msilib.View`), 1685
`fetchall()` (méthode `sqlite3.Cursor`), 429
`fetchmany()` (méthode `sqlite3.Cursor`), 429
`fetchone()` (méthode `sqlite3.Cursor`), 429
`fflags` (attribut `select.kevent`), 853
fichier binaire, 1754
fichier texte, 1763
`field_size_limit()` (dans le module `csv`), 473
`fieldnames` (attribut `csv.csvreader`), 476
`fields` (attribut `uuid.UUID`), 1171
file
 byte-code, 1659, 1745
 configuration, 478
 copying, 388
 debugger configuration, 1492
 .ini, 478
 large files, 1701
 mime.types, 1013
 modes, 16
 path configuration, 1606
 .pdbrc, 1492
 plist, 499
 temporary, 381
file ...
 compileall command line option, 1660
file (attribut `pyclbr.Class`), 1658
file (attribut `pyclbr.Function`), 1658
file control
 UNIX, 1709
file name
 temporary, 381
file object
 io module, 562
 open() built-in function, 15
--file=<file>
 trace command line option, 1509
FILE_ATTRIBUTE_ARCHIVE (dans le module `stat`), 378
FILE_ATTRIBUTE_COMPRESSED (dans le module `stat`), 378
FILE_ATTRIBUTE_DEVICE (dans le module `stat`), 378
FILE_ATTRIBUTE_DIRECTORY (dans le module `stat`), 378
FILE_ATTRIBUTE_ENCRYPTED (dans le module `stat`), 378
FILE_ATTRIBUTE_HIDDEN (dans le module `stat`), 378
FILE_ATTRIBUTE_INTEGRITY_STREAM (dans le module `stat`), 378
FILE_ATTRIBUTE_NO_SCRUB_DATA (dans le module `stat`), 378
FILE_ATTRIBUTE_NORMAL (dans le module `stat`), 378
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (dans le module `stat`), 378
FILE_ATTRIBUTE_OFFLINE (dans le module `stat`), 378
FILE_ATTRIBUTE_READONLY (dans le module `stat`), 378
FILE_ATTRIBUTE_REPARSE_POINT (dans le module `stat`), 378
FILE_ATTRIBUTE_SPARSE_FILE (dans le module `stat`), 378
FILE_ATTRIBUTE_SYSTEM (dans le module `stat`), 378
FILE_ATTRIBUTE_TEMPORARY (dans le module `stat`), 378
FILE_ATTRIBUTE_VIRTUAL (dans le module `stat`), 378
file_dispatcher (classe dans `asyncore`), 917

- `file_open()` (méthode `urllib.request.FileHandler`), 1115
- `file_size` (attribut `zipfile.ZipInfo`), 460
- `file_wrapper` (classe dans `asyncore`), 917
- `filecmp` (module), 379
- `fileConfig()` (dans le module `logging.config`), 629
- `FileCookieJar` (classe dans `http.cookiejar`), 1190
- `FileEntry` (classe dans `tkinter.tix`), 1333
- `FileExistsError`, 87
- `FileFinder` (classe dans `importlib.machinery`), 1631
- `FileHandler` (classe dans `logging`), 639
- `FileHandler` (classe dans `urllib.request`), 1109
- `FileInput` (classe dans `fileinput`), 373
- `fileinput` (module), 372
- `FileIO` (classe dans `io`), 568
- `filelineno()` (dans le module `fileinput`), 373
- `FileLoader` (classe dans `importlib.abc`), 1628
- `filemode()` (dans le module `stat`), 375
- `filename` (attribut `doctest.DocTest`), 1379
- `filename` (attribut `http.cookiejar.FileCookieJar`), 1193
- `filename` (attribut `OSError`), 84
- `filename` (attribut `traceback.TracebackException`), 1583
- `filename` (attribut `tracemalloc.Frame`), 1517
- `filename` (attribut `zipfile.ZipFile`), 457
- `filename` (attribut `zipfile.ZipInfo`), 459
- `filename()` (dans le module `fileinput`), 372
- `filename2` (attribut `OSError`), 84
- `filename_only` (dans le module `tabnanny`), 1657
- `filename_pattern` (attribut `tracemalloc.Filter`), 1516
- `filenames`
- `pathname expansion`, 385
 - `wildcard expansion`, 386
- `fileno()` (dans le module `fileinput`), 372
- `fileno()` (méthode `http.client.HTTPResponse`), 1136
- `fileno()` (méthode `io.IOBase`), 565
- `fileno()` (méthode `multiprocessing.connection.Connection`), 740
- `fileno()` (méthode `ossaudiodev.oss_audio_device`), 1238
- `fileno()` (méthode `ossaudiodev.oss_mixer_device`), 1240
- `fileno()` (méthode `select.devpoll`), 849
- `fileno()` (méthode `select.epoll`), 850
- `fileno()` (méthode `select.kqueue`), 852
- `fileno()` (méthode `selectors.DevpollSelector`), 856
- `fileno()` (méthode `selectors.EpollSelector`), 856
- `fileno()` (méthode `selectors.KqueueSelector`), 856
- `fileno()` (méthode `socketserver.BaseServer`), 1175
- `fileno()` (méthode `socket.socket`), 808
- `fileno()` (méthode `telnetlib.Telnet`), 1169
- `FileNotFoundError`, 88
- `fileobj` (attribut `selectors.SelectorKey`), 854
- `FileSelectBox` (classe dans `tkinter.tix`), 1333
- `FileType` (classe dans `argparse`), 607
- `FileWrapper` (classe dans `wsgiref.util`), 1097
- `fill()` (dans le module `textwrap`), 132
- `fill()` (méthode `textwrap.TextWrapper`), 135
- `fillcolor()` (dans le module `turtle`), 1274
- `filling()` (dans le module `turtle`), 1275
- `filter` (2to3 fixer), 1470
- `filter` (attribut `select.kevent`), 852
- `Filter` (classe dans `logging`), 620
- `Filter` (classe dans `tracemalloc`), 1516
- `filter()` (dans le module `curses`), 652
- `filter()` (dans le module `fnmatch`), 387
- `filter()` (fonction de base), 11
- `filter()` (méthode `logging.Filter`), 620
- `filter()` (méthode `logging.Handler`), 618
- `filter()` (méthode `logging.Logger`), 617
- `FILTER_DIR` (dans le module `unittest.mock`), 1442
- `filter_traces()` (méthode `tracemalloc.Snapshot`), 1517
- `filterfalse()` (dans le module `itertools`), 327
- `filterwarnings()` (dans le module `warnings`), 1562
- `finalize` (classe dans `weakref`), 234
- `find()` (dans le module `gettext`), 1245
- `find()` (méthode `bytearray`), 54
- `find()` (méthode `bytes`), 54
- `find()` (méthode `doctest.DocTestFinder`), 1380
- `find()` (méthode `mmap.mmap`), 928
- `find()` (méthode `str`), 43
- `find()` (méthode `xml.etree.ElementTree.Element`), 1040
- `find()` (méthode `xml.etree.ElementTree.ElementTree`), 1041
- `find_class()` (méthode `pickle.Unpickler`), 401
- `find_class()` (`pickle protocol`), 408
- `find_library()` (dans le module `ctypes.util`), 707
- `find_loader()` (dans le module `importlib`), 1624
- `find_loader()` (dans le module `pkgutil`), 1618
- `find_loader()` (méthode `importlib.abc.PathEntryFinder`), 1626
- `find_loader()` (méthode `importlib.machinery.FileFinder`), 1631
- `find_longest_match()` (méthode `difflib.SequenceMatcher`), 127
- `find_module()` (dans le module `imp`), 1745
- `find_module()` (méthode de la classe `importlib.machinery.PathFinder`), 1631
- `find_module()` (méthode `imp.NullImporter`), 1748
- `find_module()` (méthode `importlib.abc.Finder`), 1625
- `find_module()` (méthode `importlib.abc.MetaPathFinder`), 1625
- `find_module()` (méthode `importlib.abc.PathEntryFinder`), 1626
- `find_module()` (méthode `zipimport.zipimporter`), 1616
- `find_msvcr()` (dans le module `ctypes.util`), 707
- `find_spec()` (dans le module `importlib.util`), 1635

- `find_spec()` (méthode de la classe `importlib.machinery.PathFinder`), 1631
- `find_spec()` (méthode `importlib.abc.MetaPathFinder`), 1625
- `find_spec()` (méthode `importlib.abc.PathEntryFinder`), 1626
- `find_spec()` (méthode `importlib.machinery.FileFinder`), 1631
- `find_unused_port()` (dans le module `test.support`), 1479
- `find_user_password()` (méthode `url-lib.request.HTTPPasswordMgr`), 1113, 1114
- `findall()` (dans le module `re`), 111
- `findall()` (méthode `re.regex`), 114
- `findall()` (méthode `xml.etree.ElementTree.Element`), 1040
- `findall()` (méthode `xml.etree.ElementTree.ElementTree`), 1041
- `findCaller()` (méthode `logging.Logger`), 617
- `Finder` (classe dans `importlib.abc`), 1625
- `findfactor()` (dans le module `audioop`), 1224
- `findfile()` (dans le module `test.support`), 1476
- `findfit()` (dans le module `audioop`), 1224
- `finditer()` (dans le module `re`), 111
- `finditer()` (méthode `re.regex`), 114
- `findlabels()` (dans le module `dis`), 1665
- `findlinestarts()` (dans le module `dis`), 1665
- `findmatch()` (dans le module `mailcap`), 994
- `findmax()` (dans le module `audioop`), 1224
- `findtext()` (méthode `xml.etree.ElementTree.Element`), 1040
- `findtext()` (méthode `xml.etree.ElementTree.ElementTree`), 1041
- `finish()` (méthode `socketserver.BaseRequestHandler`), 1177
- `finish_request()` (méthode `socketserver.BaseServer`), 1176
- `firstChild` (attribut `xml.dom.Node`), 1048
- `firstkey()` (méthode `dbm.gnu.gdbm`), 417
- `firstweekday()` (dans le module `calendar`), 201
- `fix_missing_locations()` (dans le module `ast`), 1647
- `fix_sentence_endings` (attribut `text-wrap.TextWrapper`), 134
- `Flag` (classe dans `enum`), 251
- `flag_bits` (attribut `zipfile.ZipInfo`), 459
- `flags` (attribut `re.regex`), 114
- `flags` (attribut `select.kevent`), 852
- `flags` (dans le module `sys`), 1542
- `flash()` (dans le module `curses`), 652
- `flatten()` (méthode `email.generator.BytesGenerator`), 944
- `flatten()` (méthode `email.generator.Generator`), 945
- `flattening`
 - objects, 397
- `float`
 - fonction de base, 29
- `float` (classe de base), 11
- `float_info` (dans le module `sys`), 1543
- `float_repr_style` (dans le module `sys`), 1543
- floating point
 - literals, 29
 - objet, 29
- `FloatingPointError`, 83, 1609
- `FloatOperation` (classe dans `decimal`), 299
- `flock()` (dans le module `fcntl`), 1710
- `floor()` (dans le module `math`), 273
- `floor()` (in module `math`), 30
- `floordiv()` (dans le module `operator`), 343
- `flush()` (méthode `bz2.BZ2Compressor`), 447
- `flush()` (méthode `formatter.writer`), 1680
- `flush()` (méthode `io.BufferedWriter`), 570
- `flush()` (méthode `io.IOWrapper`), 565
- `flush()` (méthode `logging.Handler`), 619
- `flush()` (méthode `logging.handlers.BufferingHandler`), 647
- `flush()` (méthode `logging.handlers.MemoryHandler`), 647
- `flush()` (méthode `logging.StreamHandler`), 639
- `flush()` (méthode `lzma.LZMACompressor`), 450
- `flush()` (méthode `mailbox.Mailbox`), 998
- `flush()` (méthode `mailbox.Maildir`), 999
- `flush()` (méthode `mailbox.MH`), 1001
- `flush()` (méthode `mmap.mmap`), 928
- `flush()` (méthode `zlib.Compress`), 441
- `flush()` (méthode `zlib.Decompress`), 442
- `flush_headers()` (méthode `http.server.BaseHTTPRequestHandler`), 1184
- `flush_softspace()` (méthode `formatter.formatter`), 1678
- `flushinp()` (dans le module `curses`), 652
- `FlushKey()` (dans le module `winreg`), 1692
- `fma()` (méthode `decimal.Context`), 295
- `fma()` (méthode `decimal.Decimal`), 288
- `fmod()` (dans le module `math`), 273
- `FMT_BINARY` (dans le module `plistlib`), 501
- `FMT_XML` (dans le module `plistlib`), 500
- `fnmatch` (module), 386
- `fnmatch()` (dans le module `fnmatch`), 386
- `fnmatchcase()` (dans le module `fnmatch`), 387
- `focus()` (méthode `tkinter.ttk.Treeview`), 1325
- `fold` (attribut `datetime.datetime`), 179
- `fold` (attribut `datetime.time`), 186
- `fold()` (méthode `email.headerregistry.BaseHeader`), 954
- `fold()` (méthode `email.policy.Compat32`), 952
- `fold()` (méthode `email.policy.EmailPolicy`), 950
- `fold()` (méthode `email.policy.Policy`), 949
- `fold_binary()` (méthode `email.policy.Compat32`), 952

- `fold_binary()` (méthode `email.policy.EmailPolicy`), 950
- `fold_binary()` (méthode `email.policy.Policy`), 949
- fonction, 1757
- fonction clé, 1759
- fonction coroutine, 1755
- fonction de base
- `compile`, 78, 240, 1641
 - `complex`, 29
 - `eval`, 78, 245, 1641
 - `exec`, 10, 78, 1641
 - `float`, 29
 - `hash`, 37
 - `int`, 29
 - `len`, 35, 73
 - `max`, 35
 - `min`, 35
 - `slice`, 1673
 - `type`, 78
- fonction générique, 1757
- `FOR_ITER` (opcode), 1672
- `forget()` (dans le module `test.support`), 1476
- `forget()` (méthode `tkinter.ttk.Notebook`), 1320
- `fork()` (dans le module `os`), 553
- `fork()` (dans le module `pty`), 1708
- `ForkingMixIn` (classe dans `socketserver`), 1174
- `ForkingTCPServer` (classe dans `socketserver`), 1175
- `ForkingUDPServer` (classe dans `socketserver`), 1175
- `forkpty()` (dans le module `os`), 553
- `Form` (classe dans `tkinter.tix`), 1334
- `format` (attribut `memoryview`), 69
- `format` (attribut `struct.Struct`), 150
- `format()` (dans le module locale), 1256
- `format()` (fonction de base), 11
- `format()` (méthode `logging.Formatter`), 619
- `format()` (méthode `logging.Handler`), 619
- `format()` (méthode `pprint.PrettyPrinter`), 245
- `format()` (méthode `str`), 43
- `format()` (méthode `string.Formatter`), 92
- `format()` (méthode `traceback.StackSummary`), 1584
- `format()` (méthode `traceback.TracebackException`), 1583
- `format()` (méthode `tracemalloc.Traceback`), 1519
- `format_datetime()` (dans le module `email.utils`), 983
- `format_exc()` (dans le module `traceback`), 1582
- `format_exception()` (dans le module `traceback`), 1582
- `format_exception_only()` (dans le module `traceback`), 1582
- `format_exception_only()` (méthode `traceback.TracebackException`), 1583
- `format_field()` (méthode `string.Formatter`), 93
- `format_help()` (méthode `argparse.ArgumentParser`), 610
- `format_list()` (dans le module `traceback`), 1582
- `format_map()` (méthode `str`), 44
- `format_stack()` (dans le module `traceback`), 1582
- `format_stack_entry()` (méthode `bdb.Bdb`), 1486
- `format_string()` (dans le module locale), 1256
- `format_tb()` (dans le module `traceback`), 1582
- `format_usage()` (méthode `argparse.ArgumentParser`), 610
- `FORMAT_VALUE` (opcode), 1673
- `formataddr()` (dans le module `email.utils`), 982
- `formatargspec()` (dans le module `inspect`), 1600
- `formatargvalues()` (dans le module `inspect`), 1600
- `formatdate()` (dans le module `email.utils`), 983
- `FormatError`, 1011
- `FormatError()` (dans le module `ctypes`), 707
- `FormatException()` (méthode `logging.Formatter`), 620
- `formatmonth()` (méthode `calendar.HTMLCalendar`), 200
- `formatmonth()` (méthode `calendar.TextCalendar`), 200
- `formatStack()` (méthode `logging.Formatter`), 620
- `Formatter` (classe dans `logging`), 619
- `Formatter` (classe dans `string`), 92
- `formatter` (module), 1677
- `formatTime()` (méthode `logging.Formatter`), 620
- `formatting`
- `bytearray` (%), 62
 - `bytes` (%), 62
- `formatting, string` (%), 49
- `formatwarning()` (dans le module `warnings`), 1562
- `formatyear()` (méthode `calendar.HTMLCalendar`), 200
- `formatyear()` (méthode `calendar.TextCalendar`), 200
- `formatyearpage()` (méthode `calendar.HTMLCalendar`), 200
- `Fortran contiguous`, 1755
- `forward()` (dans le module `turtle`), 1266
- `found_terminator()` (méthode `async_chat.async_chat`), 920
- `fpathconf()` (dans le module `os`), 527
- `fpectl` (module), 1608
- `fqdn` (attribut `smtpd.SMTPChannel`), 1167
- `Fraction` (classe dans `fractions`), 306
- `fractions` (module), 306
- `frame` (attribut `tkinter.scrolledtext.ScrolledText`), 1335
- `Frame` (classe dans `tracemalloc`), 1517
- `FrameSummary` (classe dans `traceback`), 1584
- `FrameType` (dans le module `types`), 240
- `freeze_support()` (dans le module `multiprocessing`), 738
- `frexp()` (dans le module `math`), 273
- `from_address()` (méthode `ctypes._CData`), 708
- `from_buffer()` (méthode `ctypes._CData`), 708
- `from_buffer_copy()` (méthode `ctypes._CData`), 708

- `from_bytes()` (méthode de la classe `int`), 31
 - `from_callable()` (méthode de la classe `inspect.Signature`), 1597
 - `from_decimal()` (méthode `fractions.Fraction`), 307
 - `from_exception()` (méthode de la classe `traceback.TracebackException`), 1583
 - `from_file()` (méthode de la classe `zipfile.ZipInfo`), 458
 - `from_float()` (méthode `decimal.Decimal`), 288
 - `from_float()` (méthode `fractions.Fraction`), 307
 - `from_iterable()` (méthode de la classe `iter-tools.chain`), 324
 - `from_list()` (méthode de la classe `traceback.StackSummary`), 1584
 - `from_param()` (méthode `ctypes._CData`), 708
 - `from_traceback()` (méthode de la classe `dis.Bytecode`), 1663
 - `frombuf()` (méthode de la classe `tarfile.TarInfo`), 466
 - `frombytes()` (méthode `array.array`), 230
 - `fromfd()` (dans le module `socket`), 803
 - `fromfd()` (méthode `select.epoll`), 850
 - `fromfd()` (méthode `select.kqueue`), 852
 - `fromfile()` (méthode `array.array`), 230
 - `fromhex()` (méthode de la classe `bytearray`), 52
 - `fromhex()` (méthode de la classe `bytes`), 52
 - `fromhex()` (méthode de la classe `float`), 32
 - `fromkeys()` (méthode `collections.Counter`), 206
 - `fromkeys()` (méthode de la classe `dict`), 74
 - `fromlist()` (méthode `array.array`), 230
 - `fromordinal()` (méthode de la classe `datetime.date`), 174
 - `fromordinal()` (méthode de la classe `datetime.datetime`), 178
 - `fromshare()` (dans le module `socket`), 803
 - `fromstring()` (dans le module `xml.etree.ElementTree`), 1037
 - `fromstring()` (méthode `array.array`), 230
 - `fromstringlist()` (dans le module `xml.etree.ElementTree`), 1037
 - `fromtarfile()` (méthode de la classe `tarfile.TarInfo`), 466
 - `fromtimestamp()` (méthode de la classe `datetime.date`), 174
 - `fromtimestamp()` (méthode de la classe `datetime.datetime`), 177
 - `fromunicode()` (méthode `array.array`), 230
 - `fromutc()` (méthode `datetime.timezone`), 195
 - `fromutc()` (méthode `datetime.tzinfo`), 189
 - `FrozenImporter` (classe dans `importlib.machinery`), 1630
 - `FrozenSet` (classe dans `typing`), 1356
 - `frozenset` (classe de base), 71
 - `fsdecode()` (dans le module `os`), 521
 - `fsencode()` (dans le module `os`), 521
 - `fspath()` (dans le module `os`), 521
 - `fstat()` (dans le module `os`), 527
 - `fstatvfs()` (dans le module `os`), 527
 - `fsum()` (dans le module `math`), 273
 - `fsync()` (dans le module `os`), 527
 - `FTP`, 1121
 - `ftplib` (standard module), 1138
 - protocol, 1121, 1138
 - `FTP` (classe dans `ftplib`), 1139
 - `ftp_open()` (méthode `urllib.request.FTPHandler`), 1115
 - `FTP_TLS` (classe dans `ftplib`), 1139
 - `FTPHandler` (classe dans `urllib.request`), 1109
 - `ftplib` (module), 1138
 - `ftruncate()` (dans le module `os`), 527
 - `Full`, 791
 - `full()` (méthode `asyncio.Queue`), 907
 - `full()` (méthode `multiprocessing.Queue`), 737
 - `full()` (méthode `queue.Queue`), 792
 - `full_url` (attribut `urllib.request.Request`), 1109
 - `fullmatch()` (dans le module `re`), 110
 - `fullmatch()` (méthode `re.regex`), 113
 - `func` (attribut `functools.partial`), 342
 - `funcattrs` (2to3 fixer), 1470
 - `Function` (classe dans `symtable`), 1649
 - `FunctionTestCase` (classe dans `unittest`), 1403
 - `FunctionType` (dans le module `types`), 240
 - `functools` (module), 336
 - `funny_files` (attribut `filecmp.dircmp`), 380
 - `future` (2to3 fixer), 1470
 - `Future` (classe dans `asyncio`), 876
 - `Future` (classe dans `concurrent.futures`), 772
 - `FutureWarning`, 88
 - `fwalk()` (dans le module `os`), 549
- ## G
- `-g`
 - trace command line option, 1509
 - `G.722`, 1228
 - `gaierror`, 799
 - `gamma()` (dans le module `math`), 276
 - `gammavariate()` (dans le module `random`), 311
 - `garbage` (dans le module `gc`), 1590
 - `gather()` (dans le module `asyncio`), 880
 - `gather()` (méthode `curses.textpad.Textbox`), 668
 - `gauss()` (dans le module `random`), 311
 - `gc` (module), 1588
 - `gcd()` (dans le module `fractions`), 308
 - `gcd()` (dans le module `math`), 273
 - `ge()` (dans le module `operator`), 342
 - `gen_uuid()` (dans le module `msilib`), 1684
 - `générateur`, 1757
 - `générateur asynchrone`, 1754
 - `generator`, 1757
 - `Generator` (classe dans `collections.abc`), 220
 - `Generator` (classe dans `email.generator`), 944

- Generator (*classe dans typing*), 1357
 generator expression, 1757
 GeneratorExit, 83
 GeneratorType (*dans le module types*), 240
 Generic (*classe dans typing*), 1354
 generic_visit() (*méthode ast.NodeVisitor*), 1648
 genops() (*dans le module pickletools*), 1676
 gestionnaire de contexte, 1755
 gestionnaire de contexte asynchrone, 1754
 get() (*dans le module webbrowser*), 1086
 get() (*méthode asyncio.Queue*), 907
 get() (*méthode configparser.ConfigParser*), 492
 get() (*méthode dict*), 74
 get() (*méthode email.message.EmailMessage*), 935
 get() (*méthode email.message.Message*), 970
 get() (*méthode mailbox.Mailbox*), 997
 get() (*méthode multiprocessing.pool.AsyncResult*), 754
 get() (*méthode multiprocessing.Queue*), 737
 get() (*méthode multiprocessing.SimpleQueue*), 738
 get() (*méthode ossaudiodev.oss_mixer_device*), 1241
 get() (*méthode queue.Queue*), 792
 get() (*méthode tkinter.ttk.Combobox*), 1319
 get() (*méthode types.MappingProxyType*), 241
 get() (*méthode xml.etree.ElementTree.Element*), 1040
 GET_AITER (*opcode*), 1668
 get_all() (*méthode email.message.EmailMessage*), 935
 get_all() (*méthode email.message.Message*), 970
 get_all() (*méthode wsgiref.headers.Headers*), 1098
 get_all_breaks() (*méthode bdb.Bdb*), 1486
 get_all_start_methods() (*dans le module multiprocessing*), 739
 GET_ANEXT (*opcode*), 1668
 get_app() (*méthode wsgi-ref.simple_server.WSGIServer*), 1099
 get_archive_formats() (*dans le module shutil*), 393
 get_asyncgen_hooks() (*dans le module sys*), 1545
 GET_AWAITABLE (*opcode*), 1668
 get_begidx() (*dans le module readline*), 142
 get_blocking() (*dans le module os*), 527
 get_body() (*méthode email.message.EmailMessage*), 937
 get_body_encoding() (*méthode email.charset.Charset*), 980
 get_boundary() (*méthode email.message.EmailMessage*), 936
 get_boundary() (*méthode email.message.Message*), 972
 get_bpbynumber() (*méthode bdb.Bdb*), 1486
 get_break() (*méthode bdb.Bdb*), 1486
 get_breaks() (*méthode bdb.Bdb*), 1486
 get_buffer() (*méthode xdrlib.Packer*), 496
 get_buffer() (*méthode xdrlib.Unpacker*), 497
 get_bytes() (*méthode mailbox.Mailbox*), 997
 get_ca_certs() (*méthode ssl.SSLContext*), 833
 get_cache_token() (*dans le module abc*), 1579
 get_channel_binding() (*méthode ssl.SSLSocket*), 830
 get_charset() (*méthode email.message.Message*), 969
 get_charsets() (*méthode email.message.EmailMessage*), 936
 get_charsets() (*méthode email.message.Message*), 973
 get_children() (*méthode symtable.SymbolTable*), 1649
 get_children() (*méthode tkinter.ttk.Treeview*), 1325
 get_ciphers() (*méthode ssl.SSLContext*), 833
 get_clock_info() (*dans le module time*), 575
 get_close_matches() (*dans le module difflib*), 124
 get_code() (*méthode importlib.abc.InspectLoader*), 1627
 get_code() (*méthode importlib.abc.SourceLoader*), 1629
 get_code() (*méthode importlib.machinery.ExtensionFileLoader*), 1633
 get_code() (*méthode importlib.machinery.SourcelessFileLoader*), 1632
 get_code() (*méthode zipimport.zipimporter*), 1616
 get_completer() (*dans le module readline*), 142
 get_completer_delims() (*dans le module readline*), 142
 get_completion_type() (*dans le module readline*), 142
 get_config_h_filename() (*dans le module sysconfig*), 1556
 get_config_var() (*dans le module sysconfig*), 1554
 get_config_vars() (*dans le module sysconfig*), 1554
 get_content() (*dans le module email.contentmanager*), 959
 get_content() (*méthode email.contentmanager.ContentManager*), 958
 get_content() (*méthode email.message.EmailMessage*), 938
 get_content_charset() (*méthode email.message.EmailMessage*), 936
 get_content_charset() (*méthode email.message.Message*), 973
 get_content_disposition() (*méthode email.message.EmailMessage*), 937
 get_content_disposition() (*méthode email.message.Message*), 973
 get_content_maintype() (*méthode email.message.EmailMessage*), 935
 get_content_maintype() (*méthode email.message.Message*), 971
 get_content_subtype() (*méthode email.message.EmailMessage*), 935

- `get_content_subtype()` (méthode `email.message.Message`), 971
`get_content_type()` (méthode `email.message.EmailMessage`), 935
`get_content_type()` (méthode `email.message.Message`), 971
`get_context()` (dans le module `multiprocessing`), 739
`get_coroutine_wrapper()` (dans le module `sys`), 1546
`get_count()` (dans le module `gc`), 1589
`get_current_history_length()` (dans le module `readline`), 141
`get_data()` (dans le module `pkgutil`), 1619
`get_data()` (méthode `importlib.abc.FileLoader`), 1629
`get_data()` (méthode `importlib.abc.ResourceLoader`), 1627
`get_data()` (méthode `zipimport.zipimporter`), 1616
`get_date()` (méthode `mailbox.MaildirMessage`), 1004
`get_debug()` (dans le module `gc`), 1589
`get_debug()` (méthode `asyncio.AbstractEventLoop`), 866
`get_default()` (méthode `argparse.ArgumentParser`), 609
`get_default_domain()` (dans le module `nis`), 1716
`get_default_type()` (méthode `email.message.EmailMessage`), 935
`get_default_type()` (méthode `email.message.Message`), 971
`get_default_verify_paths()` (dans le module `ssl`), 822
`get_dialect()` (dans le module `csv`), 473
`get_docstring()` (dans le module `ast`), 1647
`get_doctest()` (méthode `doctest.DocTestParser`), 1380
`get_endidx()` (dans le module `readline`), 142
`get_environ()` (méthode `wsgi-ref.simple_server.WSGIRequestHandler`), 1099
`get_errno()` (dans le module `ctypes`), 707
`get_event_loop()` (dans le module `asyncio`), 870
`get_event_loop()` (méthode `asyncio.AbstractEventLoopPolicy`), 872
`get_event_loop_policy()` (dans le module `asyncio`), 872
`get_examples()` (méthode `doctest.DocTestParser`), 1380
`get_exception_handler()` (méthode `asyncio.AbstractEventLoop`), 866
`get_exec_path()` (dans le module `os`), 522
`get_extra_info()` (méthode `asyncio.BaseTransport`), 883
`get_extra_info()` (méthode `asyncio.StreamWriter`), 894
`get_field()` (méthode `string.Formatter`), 92
`get_file()` (méthode `mailbox.Babyl`), 1002
`get_file()` (méthode `mailbox.Mailbox`), 997
`get_file()` (méthode `mailbox.Maildir`), 999
`get_file()` (méthode `mailbox.mbox`), 1000
`get_file()` (méthode `mailbox.MH`), 1001
`get_file()` (méthode `mailbox.MMDF`), 1003
`get_file_breaks()` (méthode `bdb.Bdb`), 1486
`get_filename()` (méthode `email.message.EmailMessage`), 936
`get_filename()` (méthode `email.message.Message`), 972
`get_filename()` (méthode `importlib.abc.ExecutionLoader`), 1628
`get_filename()` (méthode `importlib.abc.FileLoader`), 1629
`get_filename()` (méthode `importlib.abc.ExtensionFileLoader`), 1633
`get_filename()` (méthode `zipimport.zipimporter`), 1616
`get_flags()` (méthode `mailbox.MaildirMessage`), 1004
`get_flags()` (méthode `mailbox.mboxMessage`), 1006
`get_flags()` (méthode `mailbox.MMDFMessage`), 1009
`get_folder()` (méthode `mailbox.Maildir`), 999
`get_folder()` (méthode `mailbox.MH`), 1001
`get_frees()` (méthode `symtable.Function`), 1650
`get_from()` (méthode `mailbox.mboxMessage`), 1006
`get_from()` (méthode `mailbox.MMDFMessage`), 1009
`get_full_url()` (méthode `urllib.request.Request`), 1110
`get_globals()` (méthode `symtable.Function`), 1649
`get_grouped_opcodes()` (méthode `diff-lib.SequenceMatcher`), 128
`get_handle_inheritable()` (dans le module `os`), 533
`get_header()` (méthode `urllib.request.Request`), 1110
`get_history_item()` (dans le module `readline`), 141
`get_history_length()` (dans le module `readline`), 140
`get_id()` (méthode `symtable.SymbolTable`), 1649
`get_ident()` (dans le module `_thread`), 794
`get_ident()` (dans le module `threading`), 715
`get_identifiers()` (méthode `symtable.SymbolTable`), 1649
`get_importer()` (dans le module `pkgutil`), 1618
`get_info()` (méthode `mailbox.MaildirMessage`), 1004
`get_inheritable()` (dans le module `os`), 533
`get_inheritable()` (méthode `socket.socket`), 808
`get_instructions()` (dans le module `dis`), 1665
`get_interpreter()` (dans le module `zipapp`), 1533
`GET_ITER` (opcode), 1666
`get_key()` (méthode `selectors.BaseSelector`), 855
`get_labels()` (méthode `mailbox.Babyl`), 1002
`get_labels()` (méthode `mailbox.BabylMessage`), 1008
`get_last_error()` (dans le module `ctypes`), 707
`get_line_buffer()` (dans le module `readline`), 140
`get_lineno()` (méthode `symtable.SymbolTable`), 1649

- `get_loader()` (dans le module `pkgutil`), 1618
`get_locals()` (méthode `symtable.Function`), 1649
`get_logger()` (dans le module `multiprocessing`), 758
`get_magic()` (dans le module `imp`), 1745
`get_makefile_filename()` (dans le module `sysconfig`), 1556
`get_map()` (méthode `selectors.BaseSelector`), 856
`get_matching_blocks()` (méthode `diffib.SequenceMatcher`), 127
`get_message()` (méthode `mailbox.Mailbox`), 997
`get_method()` (méthode `urllib.request.Request`), 1110
`get_methods()` (méthode `symtable.Class`), 1650
`get_mixed_type_key()` (dans le module `ipaddress`), 1221
`get_name()` (méthode `symtable.Symbol`), 1650
`get_name()` (méthode `symtable.SymbolTable`), 1649
`get_namespace()` (méthode `symtable.Symbol`), 1650
`get_namespaces()` (méthode `symtable.Symbol`), 1650
`get_nonstandard_attr()` (méthode `http.cookiejar.Cookie`), 1197
`get_nowait()` (méthode `asyncio.Queue`), 908
`get_nowait()` (méthode `multiprocessing.Queue`), 737
`get_nowait()` (méthode `queue.Queue`), 792
`get_object_traceback()` (dans le module `trace-malloc`), 1515
`get_objects()` (dans le module `gc`), 1589
`get_opcodes()` (méthode `diffib.SequenceMatcher`), 127
`get_option()` (méthode `optparse.OptionParser`), 1736
`get_option_group()` (méthode `optparse.OptionParser`), 1727
`get_osfhandle()` (dans le module `msvcrt`), 1689
`get_output_charset()` (méthode `email.charset.Charset`), 980
`get_param()` (méthode `email.message.Message`), 971
`get_parameters()` (méthode `symtable.Function`), 1649
`get_params()` (méthode `email.message.Message`), 971
`get_path()` (dans le module `sysconfig`), 1555
`get_path_names()` (dans le module `sysconfig`), 1555
`get_paths()` (dans le module `sysconfig`), 1556
`get_payload()` (méthode `email.message.Message`), 968
`get_pid()` (méthode `asyncio.BaseSubprocessTransport`), 885
`get_pipe_transport()` (méthode `asyncio.BaseSubprocessTransport`), 885
`get_platform()` (dans le module `sysconfig`), 1556
`get_poly()` (dans le module `turtle`), 1280
`get_position()` (méthode `xdrlib.Unpacker`), 497
`get_protocol()` (méthode `asyncio.BaseTransport`), 883
`get_python_version()` (dans le module `sysconfig`), 1556
`get_recsrc()` (méthode `ossaudio-dev.oss_mixer_device`), 1241
`get_referents()` (dans le module `gc`), 1589
`get_referrers()` (dans le module `gc`), 1589
`get_request()` (méthode `socketserver.BaseServer`), 1176
`get_returncode()` (méthode `asyncio.BaseSubprocessTransport`), 885
`get_scheme()` (méthode `wsgiref.handlers.BaseHandler`), 1102
`get_scheme_names()` (dans le module `sysconfig`), 1555
`get_sequences()` (méthode `mailbox.MH`), 1001
`get_sequences()` (méthode `mailbox.MHMessage`), 1007
`get_server()` (méthode `multiprocessing.managers.BaseManager`), 746
`get_server_certificate()` (dans le module `ssl`), 822
`get_shapepoly()` (dans le module `turtle`), 1278
`get_socket()` (méthode `telnetlib.Telnet`), 1169
`get_source()` (méthode `importlib.abc.InspectLoader`), 1628
`get_source()` (méthode `importlib.abc.SourceLoader`), 1629
`get_source()` (méthode `importlib.machinery.ExtensionFileLoader`), 1633
`get_source()` (méthode `importlib.machinery.SourcelessFileLoader`), 1632
`get_source()` (méthode `zipimport.zipimporter`), 1616
`get_stack()` (méthode `asyncio.Task`), 878
`get_stack()` (méthode `bdb.Bdb`), 1486
`get_start_method()` (dans le module `multiprocessing`), 739
`get_starttag_text()` (méthode `html.parser.HTMLParser`), 1025
`get_stats()` (dans le module `gc`), 1589
`get_stderr()` (méthode `wsgiref.handlers.BaseHandler`), 1102
`get_stderr()` (méthode `wsgiref.simple_server.WSGIRequestHandler`), 1100
`get_stdin()` (méthode `wsgiref.handlers.BaseHandler`), 1102
`get_string()` (méthode `mailbox.Mailbox`), 997
`get_subdir()` (méthode `mailbox.MaildirMessage`), 1004
`get_suffixes()` (dans le module `imp`), 1745
`get_symbols()` (méthode `symtable.SymbolTable`), 1649
`get_tag()` (dans le module `imp`), 1747
`get_task_factory()` (méthode `asyncio.AbstractEventLoop`), 860
`get_terminal_size()` (dans le module `os`), 532
`get_terminal_size()` (dans le module `shutil`), 395

- `get_terminator()` (méthode *asynchat.async_chat*), 920
- `get_threshold()` (dans le module *gc*), 1589
- `get_token()` (méthode *shlex.shlex*), 1299
- `get_traceback_limit()` (dans le module *tracemalloc*), 1515
- `get_traced_memory()` (dans le module *tracemalloc*), 1515
- `get_tracemalloc_memory()` (dans le module *tracemalloc*), 1515
- `get_type()` (méthode *symtable.SymbolTable*), 1649
- `get_type_hints()` (dans le module *typing*), 1359
- `get_unixfrom()` (méthode *email.message.EmailMessage*), 934
- `get_unixfrom()` (méthode *email.message.Message*), 968
- `get_unpack_formats()` (dans le module *shutil*), 394
- `get_usage()` (méthode *optparse.OptionParser*), 1738
- `get_value()` (méthode *string.Formatter*), 92
- `get_version()` (méthode *optparse.OptionParser*), 1728
- `get_visible()` (méthode *mailbox.BabylMessage*), 1008
- `get_wch()` (méthode *curses.window*), 659
- `get_write_buffer_limits()` (méthode *asyn-cio.WriteTransport*), 884
- `get_write_buffer_size()` (méthode *asyn-cio.WriteTransport*), 884
- `GET_YIELD_FROM_ITER` (opcode), 1666
- `getacl()` (méthode *imaplib.IMAP4*), 1149
- `getaddresses()` (dans le module *email.utils*), 982
- `getaddrinfo()` (dans le module *socket*), 803
- `getaddrinfo()` (méthode *asyncio.AbstractEventLoop*), 864
- `getallocatedblocks()` (dans le module *sys*), 1543
- `getannotation()` (méthode *imaplib.IMAP4*), 1149
- `getargspec()` (dans le module *inspect*), 1599
- `getargvalues()` (dans le module *inspect*), 1600
- `getatime()` (dans le module *os.path*), 369
- `getattr()` (fonction de base), 12
- `getattr_static()` (dans le module *inspect*), 1603
- `getAttribute()` (méthode *xml.dom.Element*), 1051
- `getAttributeNode()` (méthode *xml.dom.Element*), 1051
- `getAttributeNodeNS()` (méthode *xml.dom.Element*), 1051
- `getAttributeNS()` (méthode *xml.dom.Element*), 1051
- `GetBase()` (méthode *xml.parsers.expat.xmlparser*), 1075
- `getbegyx()` (méthode *curses.window*), 659
- `getbkgd()` (méthode *curses.window*), 659
- `getboolean()` (méthode *configparser.ConfigParser*), 492
- `getbuffer()` (méthode *io.BytesIO*), 569
- `getByteStream()` (méthode *xml.sax.xmlreader.InputSource*), 1073
- `getcallargs()` (dans le module *inspect*), 1600
- `getcanvas()` (dans le module *turtle*), 1286
- `getcapabilities()` (méthode *nnplib.NNTP*), 1154
- `getcaps()` (dans le module *mailcap*), 995
- `getch()` (dans le module *msvcrt*), 1690
- `getch()` (méthode *curses.window*), 659
- `getCharacterStream()` (méthode *xml.sax.xmlreader.InputSource*), 1073
- `getche()` (dans le module *msvcrt*), 1690
- `getcheckinterval()` (dans le module *sys*), 1544
- `getChild()` (méthode *logging.Logger*), 616
- `getchildren()` (méthode *xml.etree.ElementTree.Element*), 1040
- `getclasstree()` (dans le module *inspect*), 1599
- `getclosurevars()` (dans le module *inspect*), 1601
- `GetColumnInfo()` (méthode *msilib.View*), 1685
- `getColumnNumber()` (méthode *xml.sax.xmlreader.Locator*), 1072
- `getcomments()` (dans le module *inspect*), 1595
- `getcompname()` (méthode *aifc.aifc*), 1227
- `getcompname()` (méthode *sunau.AU_read*), 1230
- `getcompname()` (méthode *wave.Wave_read*), 1232
- `getcomptype()` (méthode *aifc.aifc*), 1227
- `getcomptype()` (méthode *sunau.AU_read*), 1230
- `getcomptype()` (méthode *wave.Wave_read*), 1232
- `getContentHandler()` (méthode *xml.sax.xmlreader.XMLReader*), 1071
- `getcontext()` (dans le module *decimal*), 292
- `getcoroutinelocals()` (dans le module *inspect*), 1604
- `getcoroutinestate()` (dans le module *inspect*), 1604
- `getctime()` (dans le module *os.path*), 369
- `getcwd()` (dans le module *os*), 536
- `getcwdb()` (dans le module *os*), 536
- `getcwdu (2to3 fixer)`, 1470
- `getdecoder()` (dans le module *codecs*), 152
- `getdefaultencoding()` (dans le module *sys*), 1544
- `getdefaultlocale()` (dans le module *locale*), 1255
- `getdefaulttimeout()` (dans le module *socket*), 806
- `getdlopenflags()` (dans le module *sys*), 1544
- `getdoc()` (dans le module *inspect*), 1595
- `getDOMImplementation()` (dans le module *xml.dom*), 1046
- `getDTDHandler()` (méthode *xml.sax.xmlreader.XMLReader*), 1071
- `getEffectiveLevel()` (méthode *logging.Logger*), 616
- `getegid()` (dans le module *os*), 522
- `getElementsByTagName()` (méthode *xml.dom.Document*), 1051

<code>getElementsByTagName()</code> (méthode <code>xml.dom.Element</code>), 1051	<code>getheaders()</code> (méthode <code>http.client.HTTPResponse</code>), 1136
<code>getElementsByTagNameNS()</code> (méthode <code>xml.dom.Document</code>), 1051	<code>gethostbyaddr()</code> (dans le module <code>socket</code>), 804
<code>getElementsByTagNameNS()</code> (méthode <code>xml.dom.Element</code>), 1051	<code>gethostbyaddr()</code> (in module <code>socket</code>), 525
<code>getencoder()</code> (dans le module <code>codecs</code>), 152	<code>gethostbyname()</code> (dans le module <code>socket</code>), 804
<code>getEncoding()</code> (méthode <code>xml.sax.xmlreader.InputSource</code>), 1072	<code>gethostbyname_ex()</code> (dans le module <code>socket</code>), 804
<code>getEntityResolver()</code> (méthode <code>xml.sax.xmlreader.XMLReader</code>), 1071	<code>gethostname()</code> (dans le module <code>socket</code>), 804
<code>getenv()</code> (dans le module <code>os</code>), 521	<code>gethostname()</code> (in module <code>socket</code>), 525
<code>getenvb()</code> (dans le module <code>os</code>), 521	<code>getincrementaldecoder()</code> (dans le module <code>codecs</code>), 152
<code>getErrorHandler()</code> (méthode <code>xml.sax.xmlreader.XMLReader</code>), 1071	<code>getincrementalencoder()</code> (dans le module <code>codecs</code>), 152
<code>geteuid()</code> (dans le module <code>os</code>), 522	<code>getinfo()</code> (méthode <code>zipfile.ZipFile</code>), 455
<code>getEvent()</code> (méthode <code>xml.dom.pulldom.DOMEEventStream</code>), 1061	<code>getinnerframes()</code> (dans le module <code>inspect</code>), 1602
<code>getEventCategory()</code> (méthode <code>logging.handlers.NTEventLogHandler</code>), 646	<code>GetInputContext()</code> (méthode <code>xml.parsers.expat.xmlparser</code>), 1075
<code>getEventType()</code> (méthode <code>logging.handlers.NTEventLogHandler</code>), 646	<code>getint()</code> (méthode <code>configparser.ConfigParser</code>), 492
<code>getException()</code> (méthode <code>xml.sax.SAXException</code>), 1064	<code>GetInteger()</code> (méthode <code>msilib.Record</code>), 1686
<code>getFeature()</code> (méthode <code>xml.sax.xmlreader.XMLReader</code>), 1071	<code>getitem()</code> (dans le module <code>operator</code>), 345
<code>GetFieldCount()</code> (méthode <code>msilib.Record</code>), 1686	<code>getiterator()</code> (méthode <code>xml.etree.ElementTree.Element</code>), 1040
<code>getfile()</code> (dans le module <code>inspect</code>), 1595	<code>getiterator()</code> (méthode <code>xml.etree.ElementTree.ElementTree</code>), 1041
<code>getfilesystemencodeerrors()</code> (dans le module <code>sys</code>), 1544	<code>getitimer()</code> (dans le module <code>signal</code>), 924
<code>getfilesystemencoding()</code> (dans le module <code>sys</code>), 1544	<code>getkey()</code> (méthode <code>curses.window</code>), 659
<code>getfirst()</code> (méthode <code>cgi.FieldStorage</code>), 1091	<code>GetLastError()</code> (dans le module <code>ctypes</code>), 707
<code>getfloat()</code> (méthode <code>configparser.ConfigParser</code>), 492	<code>getLength()</code> (méthode <code>xml.sax.xmlreader.Attributes</code>), 1073
<code>getfmts()</code> (méthode <code>ossaudiodev.oss_audio_device</code>), 1239	<code>getLevelName()</code> (dans le module <code>logging</code>), 626
<code>getfqdn()</code> (dans le module <code>socket</code>), 804	<code>getline()</code> (dans le module <code>linecache</code>), 387
<code>getframeinfo()</code> (dans le module <code>inspect</code>), 1602	<code>getLineNumber()</code> (méthode <code>xml.sax.xmlreader.Locator</code>), 1072
<code>getframerate()</code> (méthode <code>aifc.aifc</code>), 1227	<code>getlist()</code> (méthode <code>cgi.FieldStorage</code>), 1091
<code>getframerate()</code> (méthode <code>sunau.AU_read</code>), 1230	<code>getloadavg()</code> (dans le module <code>os</code>), 560
<code>getframerate()</code> (méthode <code>wave.Wave_read</code>), 1232	<code>getlocale()</code> (dans le module <code>locale</code>), 1255
<code>getfullargspec()</code> (dans le module <code>inspect</code>), 1599	<code>getLogger()</code> (dans le module <code>logging</code>), 624
<code>getgeneratorlocals()</code> (dans le module <code>inspect</code>), 1604	<code>getLoggerClass()</code> (dans le module <code>logging</code>), 624
<code>getgeneratorstate()</code> (dans le module <code>inspect</code>), 1603	<code>getlogin()</code> (dans le module <code>os</code>), 522
<code>getgid()</code> (dans le module <code>os</code>), 522	<code>getLogRecordFactory()</code> (dans le module <code>logging</code>), 624
<code>getgrall()</code> (dans le module <code>grp</code>), 1704	<code>getmark()</code> (méthode <code>aifc.aifc</code>), 1227
<code>getgrgid()</code> (dans le module <code>grp</code>), 1704	<code>getmark()</code> (méthode <code>sunau.AU_read</code>), 1230
<code>getgrnam()</code> (dans le module <code>grp</code>), 1704	<code>getmark()</code> (méthode <code>wave.Wave_read</code>), 1232
<code>getgrouplist()</code> (dans le module <code>os</code>), 522	<code>getmarkers()</code> (méthode <code>aifc.aifc</code>), 1227
<code>getgroups()</code> (dans le module <code>os</code>), 522	<code>getmarkers()</code> (méthode <code>sunau.AU_read</code>), 1230
<code>getheader()</code> (méthode <code>http.client.HTTPResponse</code>), 1136	<code>getmarkers()</code> (méthode <code>wave.Wave_read</code>), 1232
	<code>getmaxyx()</code> (méthode <code>curses.window</code>), 659
	<code>getmember()</code> (méthode <code>tarfile.TarFile</code>), 464
	<code>getmembers()</code> (dans le module <code>inspect</code>), 1592
	<code>getmembers()</code> (méthode <code>tarfile.TarFile</code>), 464
	<code>getMessage()</code> (méthode <code>logging.LogRecord</code>), 621
	<code>getMessage()</code> (méthode <code>xml.sax.SAXException</code>), 1064
	<code>getMessageID()</code> (méthode <code>logging.handlers.NTEventLogHandler</code>), 646

- `getmodule()` (dans le module *inspect*), 1595
`getmodulename()` (dans le module *inspect*), 1593
`getmouse()` (dans le module *curses*), 652
`getmro()` (dans le module *inspect*), 1600
`getmtime()` (dans le module *os.path*), 369
`getname()` (méthode *chunk.Chunk*), 1234
`getName()` (méthode *threading.Thread*), 718
`GetNameByQName()` (méthode *xml.sax.xmlreader.AttributesNS*), 1073
`getnameinfo()` (dans le module *socket*), 804
`getnameinfo()` (méthode *asyncio.AbstractEventLoop*), 864
`getnames()` (méthode *tarfile.TarFile*), 464
`getNames()` (méthode *xml.sax.xmlreader.Attributes*), 1073
`getnchannels()` (méthode *aifc.aifc*), 1227
`getnchannels()` (méthode *sunau.AU_read*), 1230
`getnchannels()` (méthode *wave.Wave_read*), 1232
`getnframes()` (méthode *aifc.aifc*), 1227
`getnframes()` (méthode *sunau.AU_read*), 1230
`getnframes()` (méthode *wave.Wave_read*), 1232
`getnode`, 1172
`getnode()` (dans le module *uuid*), 1171
`getopt` (module), 612
`getopt()` (dans le module *getopt*), 612
`GetoptError`, 612
`getouterframes()` (dans le module *inspect*), 1602
`getoutput()` (dans le module *subprocess*), 788
`getpagesize()` (dans le module *resource*), 1715
`getparams()` (méthode *aifc.aifc*), 1227
`getparams()` (méthode *sunau.AU_read*), 1230
`getparams()` (méthode *wave.Wave_read*), 1232
`getparyx()` (méthode *curses.window*), 659
`getpass` (module), 650
`getpass()` (dans le module *getpass*), 650
`GetPassWarning`, 650
`getpeercert()` (méthode *ssl.SSLSocket*), 829
`getpeername()` (méthode *socket.socket*), 808
`getpen()` (dans le module *turtle*), 1280
`getpgid()` (dans le module *os*), 522
`getpgrp()` (dans le module *os*), 522
`getpid()` (dans le module *os*), 523
`getpos()` (méthode *html.parser.HTMLParser*), 1025
`getppid()` (dans le module *os*), 523
`getpreferredencoding()` (dans le module *locale*), 1255
`getpriority()` (dans le module *os*), 523
`getprofile()` (dans le module *sys*), 1545
`GetProperty()` (méthode *msilib.SummaryInformation*), 1685
`GetProperty()` (méthode *xml.sax.xmlreader.XMLReader*), 1071
`GetPropertyCount()` (méthode *msilib.SummaryInformation*), 1685
`getprotobyname()` (dans le module *socket*), 804
`getproxies()` (dans le module *urllib.request*), 1106
`getPublicId()` (méthode *xml.sax.xmlreader.InputSource*), 1072
`getPublicId()` (méthode *xml.sax.xmlreader.Locator*), 1072
`getpwall()` (dans le module *pwd*), 1703
`getpwnam()` (dans le module *pwd*), 1703
`getpwuid()` (dans le module *pwd*), 1703
`getQNameByName()` (méthode *xml.sax.xmlreader.AttributesNS*), 1073
`getQNames()` (méthode *xml.sax.xmlreader.AttributesNS*), 1073
`getquota()` (méthode *imaplib.IMAP4*), 1149
`getquotaroot()` (méthode *imaplib.IMAP4*), 1149
`getrandbits()` (dans le module *random*), 309
`getrandom()` (dans le module *os*), 561
`getreader()` (dans le module *codecs*), 152
`getrecursionlimit()` (dans le module *sys*), 1544
`getrefcount()` (dans le module *sys*), 1544
`getresgid()` (dans le module *os*), 523
`getresponse()` (méthode *http.client.HTTPConnection*), 1135
`getresuid()` (dans le module *os*), 523
`getrlimit()` (dans le module *resource*), 1713
`getroot()` (méthode *xml.etree.ElementTree.ElementTree*), 1041
`getrusage()` (dans le module *resource*), 1715
`getsample()` (dans le module *audioop*), 1224
`getsampwidth()` (méthode *aifc.aifc*), 1227
`getsampwidth()` (méthode *sunau.AU_read*), 1230
`getsampwidth()` (méthode *wave.Wave_read*), 1232
`getscreen()` (dans le module *turtle*), 1280
`getservbyname()` (dans le module *socket*), 804
`getservbyport()` (dans le module *socket*), 804
`GetSetDescriptorType` (dans le module *types*), 241
`getshapes()` (dans le module *turtle*), 1286
`getsid()` (dans le module *os*), 525
`getsignal()` (dans le module *signal*), 923
`getsitpackages()` (dans le module *site*), 1607
`getsize()` (dans le module *os.path*), 369
`getsize()` (méthode *chunk.Chunk*), 1234
`getsizeof()` (dans le module *sys*), 1544
`getsockname()` (méthode *socket.socket*), 808
`getsockopt()` (méthode *socket.socket*), 808
`getsource()` (dans le module *inspect*), 1595
`getsourcefile()` (dans le module *inspect*), 1595
`getsourcelines()` (dans le module *inspect*), 1595
`getspall()` (dans le module *spwd*), 1703
`getspnam()` (dans le module *spwd*), 1703
`getstate()` (dans le module *random*), 309
`getstate()` (méthode *codecs.IncrementalDecoder*), 157
`getstate()` (méthode *codecs.IncrementalEncoder*), 156
`getstatusoutput()` (dans le module *subprocess*), 788

- `getstr()` (méthode *curses.window*), 659
`GetString()` (méthode *msilib.Record*), 1686
`getSubject()` (méthode *logging.handlers.SMTPHandler*), 646
`GetSummaryInformation()` (méthode *msilib.Database*), 1684
`getswitchinterval()` (dans le module *sys*), 1545
`getSystemId()` (méthode *xml.sax.xmlreader.InputSource*), 1072
`getSystemId()` (méthode *xml.sax.xmlreader.Locator*), 1072
`getsyx()` (dans le module *curses*), 652
`gettarinfo()` (méthode *tarfile.TarFile*), 465
`gettempdir()` (dans le module *tempfile*), 383
`gettempdirb()` (dans le module *tempfile*), 383
`gettempprefix()` (dans le module *tempfile*), 383
`gettempprefixb()` (dans le module *tempfile*), 383
`getTestCaseNames()` (méthode *unittest.TestLoader*), 1406
`gettext` (module), 1243
`gettext()` (dans le module *gettext*), 1244
`gettext()` (dans le module locale), 1258
`gettext()` (méthode *gettext.GNUTranslations*), 1247
`gettext()` (méthode *gettext.NullTranslations*), 1246
`gettimeout()` (méthode *socket.socket*), 808
`gettrace()` (dans le module *sys*), 1545
`getturtle()` (dans le module *turtle*), 1280
`getType()` (méthode *xml.sax.xmlreader.Attributes*), 1073
`getuid()` (dans le module *os*), 523
`geturl()` (méthode *urllib.parse.urllib.parse.SplitResult*), 1126
`getuser()` (dans le module *getpass*), 650
`getuserbase()` (dans le module *site*), 1607
`getusersitepackages()` (dans le module *site*), 1607
`getvalue()` (méthode *io.BytesIO*), 569
`getvalue()` (méthode *io.StringIO*), 572
`getValue()` (méthode *xml.sax.xmlreader.Attributes*), 1073
`getValueByQName()` (méthode *xml.sax.xmlreader.AttributesNS*), 1073
`getwch()` (dans le module *msvcrt*), 1690
`getwche()` (dans le module *msvcrt*), 1690
`getweakrefcount()` (dans le module *weakref*), 233
`getweakrefs()` (dans le module *weakref*), 233
`getwelcome()` (méthode *ftplib.FTP*), 1140
`getwelcome()` (méthode *nntplib.NNTP*), 1154
`getwelcome()` (méthode *poplib.POP3*), 1144
`getwin()` (dans le module *curses*), 653
`getwindowsversion()` (dans le module *sys*), 1545
`getwriter()` (dans le module *codecs*), 152
`getxattr()` (dans le module *os*), 549
`getyx()` (méthode *curses.window*), 660
`gid` (attribut *tarfile.TarInfo*), 466
`GIL`, 1757
`glob`
 module, 386
`glob` (module), 385
`glob()` (dans le module *glob*), 385
`glob()` (méthode *msilib.Directory*), 1687
`glob()` (méthode *pathlib.Path*), 363
`globals()` (fonction de base), 12
`globals` (attribut *doctest.DocTest*), 1378
`gmtime()` (dans le module *time*), 575
`gname` (attribut *tarfile.TarInfo*), 466
`GNOME`, 1248
`GNU_FORMAT` (dans le module *tarfile*), 462
`gnu_getopt()` (dans le module *getopt*), 612
`GNUTranslations` (classe dans *gettext*), 1247
`got` (attribut *doctest.DocTestFailure*), 1385
`goto()` (dans le module *turtle*), 1266
Graphical User Interface, 1303
`GREATER` (dans le module *token*), 1651
`GREATEREQUAL` (dans le module *token*), 1651
Greenwich Mean Time, 574
`GRND_NONBLOCK` (dans le module *os*), 562
`GRND_RANDOM` (dans le module *os*), 562
`Group` (classe dans *email.headerregistry*), 957
`group()` (méthode *nntplib.NNTP*), 1156
`group()` (méthode *pathlib.Path*), 363
`group()` (méthode *re.match*), 114
`groupby()` (dans le module *itertools*), 327
`groupdict()` (méthode *re.match*), 116
`groupindex` (attribut *re.regex*), 114
`groups` (attribut *email.headerregistry.AddressHeader*), 955
`groups` (attribut *re.regex*), 114
`groups()` (méthode *re.match*), 115
`grp` (module), 1704
`gt()` (dans le module *operator*), 342
`guess_all_extensions()` (dans le module *mimetypes*), 1013
`guess_all_extensions()` (méthode *mimetypes.MimeTypes*), 1014
`guess_extension()` (dans le module *mimetypes*), 1013
`guess_extension()` (méthode *mimetypes.MimeTypes*), 1014
`guess_scheme()` (dans le module *wsgiref.util*), 1096
`guess_type()` (dans le module *mimetypes*), 1012
`guess_type()` (méthode *mimetypes.MimeTypes*), 1014
GUI, 1303
`gzip` (module), 443
`GzipFile` (classe dans *gzip*), 443
- ## H
- `-h`
 json.tool command line option, 994

- timeit command line option, 1506
- tokenize command line option, 1655
- zipapp command line option, 1532
- hachable, 1758
- halfdelay() (dans le module curses), 653
- Handle (classe dans asyncio), 867
- handle() (méthode http.server.BaseHTTPRequestHandler), 1183
- handle() (méthode logging.Handler), 619
- handle() (méthode logging.handlers.QueueListener), 649
- handle() (méthode logging.Logger), 617
- handle() (méthode logging.NullHandler), 639
- handle() (méthode socketserver.BaseRequestHandler), 1177
- handle() (méthode wsgi-ref.simple_server.WSGIRequestHandler), 1100
- handle_accept() (méthode asyncore.dispatcher), 916
- handle_accepted() (méthode asyncore.dispatcher), 916
- handle_charref() (méthode html.parser.HTMLParser), 1025
- handle_close() (méthode asyncore.dispatcher), 916
- handle_comment() (méthode html.parser.HTMLParser), 1025
- handle_connect() (méthode asyncore.dispatcher), 916
- handle_data() (méthode html.parser.HTMLParser), 1025
- handle_decl() (méthode html.parser.HTMLParser), 1026
- handle_defect() (méthode email.policy.Policy), 948
- handle_endtag() (méthode html.parser.HTMLParser), 1025
- handle_entityref() (méthode html.parser.HTMLParser), 1025
- handle_error() (méthode asyncore.dispatcher), 916
- handle_error() (méthode socketserver.BaseServer), 1176
- handle_expect_100() (méthode http.server.BaseHTTPRequestHandler), 1183
- handle_expt() (méthode asyncore.dispatcher), 916
- handle_one_request() (méthode http.server.BaseHTTPRequestHandler), 1183
- handle_pi() (méthode html.parser.HTMLParser), 1026
- handle_read() (méthode asyncore.dispatcher), 916
- handle_request() (méthode socketserver.BaseServer), 1175
- handle_request() (méthode xmlrpc.server.CGIXMLRPCRequestHandler), 1209
- handle_startendtag() (méthode html.parser.HTMLParser), 1025
- handle_starttag() (méthode html.parser.HTMLParser), 1025
- handle_timeout() (méthode socketserver.BaseServer), 1176
- handle_write() (méthode asyncore.dispatcher), 916
- handleError() (méthode logging.Handler), 619
- handleError() (méthode logging.handlers.SocketHandler), 643
- Handler (classe dans logging), 618
- handler() (dans le module cgiib), 1095
- harmonic_mean() (dans le module statistics), 316
- HAS_ALPN (dans le module ssl), 827
- has_children() (méthode symtable.SymbolTable), 1649
- has_colors() (dans le module curses), 653
- HAS_ECDH (dans le module ssl), 827
- has_exec() (méthode symtable.SymbolTable), 1649
- has_extn() (méthode smtplib.SMTP), 1161
- has_header() (méthode csv.Sniffer), 474
- has_header() (méthode urllib.request.Request), 1110
- has_ic() (dans le module curses), 653
- has_il() (dans le module curses), 653
- has_ipv6 (dans le module socket), 802
- has_key (2to3 fixer), 1470
- has_key() (dans le module curses), 653
- has_location (attribut import-lib.machinery.ModuleSpec), 1633
- has_nonstandard_attr() (méthode http.cookiejar.Cookie), 1197
- HAS_NPN (dans le module ssl), 827
- has_option() (méthode configparser.ConfigParser), 491
- has_option() (méthode optparse.OptionParser), 1736
- has_section() (méthode configparser.ConfigParser), 491
- HAS_SNI (dans le module ssl), 827
- has_ticket (attribut ssl.SSLSession), 845
- HAS_TLSv1_3 (dans le module ssl), 827
- hasattr() (fonction de base), 12
- hasAttribute() (méthode xml.dom.Element), 1051
- hasAttributeNS() (méthode xml.dom.Element), 1051
- hasAttributes() (méthode xml.dom.Node), 1048
- hasChildNodes() (méthode xml.dom.Node), 1048
- hascompare (dans le module dis), 1674
- hasconst (dans le module dis), 1674
- hasFeature() (méthode xml.dom.DOMImplementation), 1047
- hasfree (dans le module dis), 1674
- hash
 - fonction de base, 37
- hash() (fonction de base), 12
- hash_info (dans le module sys), 1546
- Hashable (classe dans collections.abc), 219

- Hashable (classe dans *typing*), 1355
- hasHandlers() (méthode *logging.Logger*), 617
- hash.block_size (dans le module *hashlib*), 505
- hash.digest_size (dans le module *hashlib*), 505
- hashlib (module), 503
- hasjabs (dans le module *dis*), 1674
- hasjrel (dans le module *dis*), 1674
- haslocal (dans le module *dis*), 1674
- hasname (dans le module *dis*), 1674
- HAVE_ARGUMENT (opcode), 1674
- HAVE_THREADS (dans le module *decimal*), 297
- HCI_DATA_DIR (dans le module *socket*), 802
- HCI_FILTER (dans le module *socket*), 802
- HCI_TIME_STAMP (dans le module *socket*), 802
- head() (méthode *nntplib.NNTP*), 1157
- Header (classe dans *email.header*), 977
- header_encode() (méthode *email.charset.Charset*), 980
- header_encode_lines() (méthode *email.charset.Charset*), 980
- header_encoding (attribut *email.charset.Charset*), 979
- header_factory (attribut *email.policy.EmailPolicy*), 950
- header_fetch_parse() (méthode *email.policy.Compat32*), 951
- header_fetch_parse() (méthode *email.policy.EmailPolicy*), 950
- header_fetch_parse() (méthode *email.policy.Policy*), 949
- header_items() (méthode *urllib.request.Request*), 1110
- header_max_count() (méthode *email.policy.EmailPolicy*), 950
- header_max_count() (méthode *email.policy.Policy*), 948
- header_offset (attribut *zipfile.ZipInfo*), 460
- header_source_parse() (méthode *email.policy.Compat32*), 951
- header_source_parse() (méthode *email.policy.EmailPolicy*), 950
- header_source_parse() (méthode *email.policy.Policy*), 948
- header_store_parse() (méthode *email.policy.Compat32*), 951
- header_store_parse() (méthode *email.policy.EmailPolicy*), 950
- header_store_parse() (méthode *email.policy.Policy*), 949
- HeaderError, 462
- HeaderParseError, 952
- HeaderParser (classe dans *email.parser*), 942
- HeaderRegistry (classe dans *email.headerregistry*), 956
- headers
 - MIME, 1012, 1088
- headers (attribut *http.server.BaseHTTPRequestHandler*), 1182
- headers (attribut *urllib.error.HTTPError*), 1129
- headers (attribut *xmlrpc.client.ProtocolError*), 1203
- Headers (classe dans *wsgiref.headers*), 1098
- heading() (dans le module *turtle*), 1271
- heading() (méthode *tkinter.ttk.Treeview*), 1325
- heapify() (dans le module *heapq*), 223
- heapmin() (dans le module *msvcrt*), 1690
- heappop() (dans le module *heapq*), 222
- heappush() (dans le module *heapq*), 222
- heappushpop() (dans le module *heapq*), 223
- heapq (module), 222
- heapreplace() (dans le module *heapq*), 223
- helo() (méthode *smtplib.SMTP*), 1161
- help
 - online, 1362
- help
 - json.tool command line option, 994
 - timeit command line option, 1506
 - tokenize command line option, 1655
 - trace command line option, 1508
 - zipapp command line option, 1532
- help (attribut *optparse.Option*), 1732
- help (pdb command), 1492
- help() (fonction de base), 12
- help() (méthode *nntplib.NNTP*), 1157
- herror, 799
- hex (attribut *uuid.UUID*), 1171
- hex() (fonction de base), 12
- hex() (méthode *bytearray*), 53
- hex() (méthode *bytes*), 52
- hex() (méthode *float*), 32
- hex() (méthode *memoryview*), 66
- hexadecimal
 - literals, 29
- hexbin() (dans le module *binhex*), 1018
- hexdigest() (méthode *hashlib.hash*), 505
- hexdigest() (méthode *hashlib.shake*), 506
- hexdigest() (méthode *hmac.HMAC*), 514
- hexdigits (dans le module *string*), 92
- hexlify() (dans le module *binascii*), 1020
- hexversion (dans le module *sys*), 1546
- hidden() (méthode *curses.panel.Panel*), 671
- hide() (méthode *curses.panel.Panel*), 671
- hide() (méthode *tkinter.ttk.Notebook*), 1320
- hide_cookie2 (attribut *http.cookiejar.CookiePolicy*), 1194
- hideturtle() (dans le module *turtle*), 1276
- HierarchyRequestErr, 1053
- HIGHEST_PROTOCOL (dans le module *pickle*), 399
- HKEY_CLASSES_ROOT (dans le module *winreg*), 1695

- HKEY_CURRENT_CONFIG (dans le module winreg), 1695
HKEY_CURRENT_USER (dans le module winreg), 1695
HKEY_DYN_DATA (dans le module winreg), 1695
HKEY_LOCAL_MACHINE (dans le module winreg), 1695
HKEY_PERFORMANCE_DATA (dans le module winreg), 1695
HKEY_USERS (dans le module winreg), 1695
hline() (méthode curses.window), 660
HList (classe dans tkinter.tix), 1333
hls_to_rgb() (dans le module colorsys), 1235
hmac (module), 514
HOME, 369
home() (dans le module turtle), 1267
home() (méthode de la classe pathlib.Path), 362
HOMEDRIVE, 369
HOMEPATH, 369
hook_compressed() (dans le module fileinput), 373
hook_encoded() (dans le module fileinput), 374
host (attribut urllib.request.Request), 1109
hostmask (attribut ipaddress.IPv4Network), 1216
hostmask (attribut ipaddress.IPv6Network), 1218
hosts (attribut netrc.netrc), 495
hosts() (méthode ipaddress.IPv4Network), 1216
hosts() (méthode ipaddress.IPv6Network), 1218
hour (attribut datetime.datetime), 179
hour (attribut datetime.time), 185
HRESULT (classe dans ctypes), 711
hStdError (attribut subprocess.STARTUPINFO), 782
hStdInput (attribut subprocess.STARTUPINFO), 782
hStdOutput (attribut subprocess.STARTUPINFO), 782
hsv_to_rgb() (dans le module colorsys), 1235
ht() (dans le module turtle), 1276
HTML, 1024, 1121
html (module), 1023
html5 (dans le module html.entities), 1028
HTMLCalendar (classe dans calendar), 200
HtmlDiff (classe dans difflib), 123
html.entities (module), 1028
HTMLParser (classe dans html.parser), 1024
html.parser (module), 1024
htonl() (dans le module socket), 805
htons() (dans le module socket), 805
HTTP
 http (standard module), 1130
 http.client (standard module), 1132
 protocol, 1088, 1121, 1130, 1132, 1181
HTTP (dans le module email.policy), 951
http (module), 1130
http_error_301() (méthode url-
 lib.request.HTTPRedirectHandler), 1113
http_error_302() (méthode url-
 lib.request.HTTPRedirectHandler), 1113
http_error_303() (méthode url-
 lib.request.HTTPRedirectHandler), 1113
http_error_307() (méthode url-
 lib.request.HTTPRedirectHandler), 1113
http_error_401() (méthode url-
 lib.request.HTTPBasicAuthHandler), 1114
http_error_401() (méthode url-
 lib.request.HTTPDigestAuthHandler), 1115
http_error_407() (méthode url-
 lib.request.ProxyBasicAuthHandler), 1114
http_error_407() (méthode url-
 lib.request.ProxyDigestAuthHandler), 1115
http_error_auth_reged() (méthode url-
 lib.request.AbstractBasicAuthHandler), 1114
http_error_auth_reged() (méthode url-
 lib.request.AbstractDigestAuthHandler), 1114
http_error_default() (méthode url-
 lib.request.BaseHandler), 1112
http_error_nnn() (méthode url-
 lib.request.BaseHandler), 1112
http_open() (méthode urllib.request.HTTPHandler),
 1115
HTTP_PORT (dans le module http.client), 1134
http_proxy, 1105, 1118
http_response() (méthode url-
 lib.request.HTTPErrorProcessor), 1116
http_version (attribut wsgiref.handlers.BaseHandler),
 1103
HTTPBasicAuthHandler (classe dans urllib.request),
 1108
http.client (module), 1132
HTTPConnection (classe dans http.client), 1132
http.cookiejar (module), 1190
HTTPCookieProcessor (classe dans urllib.request),
 1107
http.cookies (module), 1186
httpd, 1181
HTTPDefaultErrorHandler (classe dans url-
 lib.request), 1107
HTTPDigestAuthHandler (classe dans url-
 lib.request), 1108
HTTPError, 1128
HTTPErrorProcessor (classe dans urllib.request),
 1109
HTTPException, 1133
HTTPHandler (classe dans logging.handlers), 647
HTTPHandler (classe dans urllib.request), 1108
HTTPPasswordMgr (classe dans urllib.request), 1107
HTTPPasswordMgrWithDefaultRealm (classe
 dans urllib.request), 1108
HTTPPasswordMgrWithPriorAuth (classe dans
 urllib.request), 1108
HTTPRedirectHandler (classe dans urllib.request),
 1107

- [HTTPResponse \(classe dans `http.client`\)](#), 1133
[https_open\(\) \(méthode `urllib.request.HTTPSHandler`\)](#), 1115
[HTTPS_PORT \(dans le module `http.client`\)](#), 1134
[https_response\(\) \(méthode `urllib.request.HTTPErrorProcessor`\)](#), 1116
[HTTPSConnection \(classe dans `http.client`\)](#), 1133
[HTTPServer \(classe dans `http.server`\)](#), 1181
[http.server \(module\)](#), 1181
[HTTPSHandler \(classe dans `urllib.request`\)](#), 1108
[HTTPStatus \(classe dans `http`\)](#), 1130
[hypot\(\) \(dans le module `math`\)](#), 275
- I**
- [I \(dans le module `re`\)](#), 109
-i list
 compileall command line option, 1660
[I/O control](#)
 buffering, 17, 808
 POSIX, 1706
 tty, 1706
 UNIX, 1709
[iadd\(\) \(dans le module `operator`\)](#), 348
[iand\(\) \(dans le module `operator`\)](#), 348
[iconcat\(\) \(dans le module `operator`\)](#), 348
[id \(attribut `ssl.SSLSession`\)](#), 845
[id\(\) \(fonction de base\)](#), 13
[id\(\) \(méthode `unittest.TestCase`\)](#), 1403
[idcok\(\) \(méthode `curses.window`\)](#), 660
[ident \(attribut `select.kevent`\)](#), 852
[ident \(attribut `threading.Thread`\)](#), 718
[identchars \(attribut `cmd.Cmd`\)](#), 1295
[identify\(\) \(méthode `tkinter.ttk.Notebook`\)](#), 1320
[identify\(\) \(méthode `tkinter.ttk.Treeview`\)](#), 1325
[identify\(\) \(méthode `tkinter.ttk.Widget`\)](#), 1318
[identify_column\(\) \(méthode `tkinter.ttk.Treeview`\)](#), 1326
[identify_element\(\) \(méthode `tkinter.ttk.Treeview`\)](#), 1326
[identify_region\(\) \(méthode `tkinter.ttk.Treeview`\)](#), 1326
[identify_row\(\) \(méthode `tkinter.ttk.Treeview`\)](#), 1326
[idioms \(2to3 fixer\)](#), 1470
[IDLE](#), 1336, 1758
[IDLE*STARTUP](#), 1342
[idlok\(\) \(méthode `curses.window`\)](#), 660
[IEEE-754](#), 1608
[if](#)
 état, 27
[if_indextoname\(\) \(dans le module `socket`\)](#), 806
[if_nameindex\(\) \(dans le module `socket`\)](#), 806
[if_nametoindex\(\) \(dans le module `socket`\)](#), 806
[ifloordiv\(\) \(dans le module `operator`\)](#), 348
[iglob\(\) \(dans le module `glob`\)](#), 385
[ignorableWhitespace\(\) \(méthode `xml.sax.handler.ContentHandler`\)](#), 1067
[ignore \(pdb command\)](#), 1492
[ignore_errors\(\) \(dans le module `codecs`\)](#), 155
[IGNORE_EXCEPTION_DETAIL \(dans le module `doctest`\)](#), 1371
[ignore_patterns\(\) \(dans le module `shutil`\)](#), 390
[IGNORECASE \(dans le module `re`\)](#), 109
--ignore-dir=<dir>
 trace command line option, 1509
--ignore-module=<mod>
 trace command line option, 1509
[ihave\(\) \(méthode `nntplib.NNTP`\)](#), 1157
[IISCGIHandler \(classe dans `wsgiref.handlers`\)](#), 1101
[ilshift\(\) \(dans le module `operator`\)](#), 348
[imag \(attribut `numbers.Complex`\)](#), 269
[imap\(\) \(méthode `multiprocessing.pool.Pool`\)](#), 753
[IMAP4](#)
 protocol, 1146
[IMAP4 \(classe dans `imaplib`\)](#), 1146
[IMAP4_SSL](#)
 protocol, 1146
[IMAP4_SSL \(classe dans `imaplib`\)](#), 1146
[IMAP4_stream](#)
 protocol, 1146
[IMAP4_stream \(classe dans `imaplib`\)](#), 1147
[IMAP4.abort](#), 1146
[IMAP4.error](#), 1146
[IMAP4.readonly](#), 1146
[imap_unordered\(\) \(méthode `multiprocessing.pool.Pool`\)](#), 753
[imaplib \(module\)](#), 1146
[imatmul\(\) \(dans le module `operator`\)](#), 348
[imgchr \(module\)](#), 1236
[immedok\(\) \(méthode `curses.window`\)](#), 660
[immutable](#), 1758
[immutable](#)
 sequence types, 37
[imod\(\) \(dans le module `operator`\)](#), 348
[imp](#)
 module, 23
[imp \(module\)](#), 1745
[ImpImporter \(classe dans `pkgutil`\)](#), 1617
[implementation \(dans le module `sys`\)](#), 1546
[ImpLoader \(classe dans `pkgutil`\)](#), 1618
[import](#)
 état, 23, 1606, 1745
[import \(2to3 fixer\)](#), 1471
[import_fresh_module\(\) \(dans le module `test.support`\)](#), 1479
[IMPORT_FROM \(opcode\)](#), 1671
[import_module\(\) \(dans le module `importlib`\)](#), 1623
[import_module\(\) \(dans le module `test.support`\)](#), 1478
[IMPORT_NAME \(opcode\)](#), 1671

- IMPORT_STAR (*opcode*), 1669
- importateur, 1758
- importer, 1758
- ImportError, 83
- importlib (*module*), 1623
- importlib.abc (*module*), 1625
- importlib.machinery (*module*), 1630
- importlib.util (*module*), 1634
- imports (*2to3 fixer*), 1471
- imports2 (*2to3 fixer*), 1471
- ImportWarning, 88
- ImproperConnectionState, 1133
- imul () (*dans le module operator*), 348
- in
 - opérateur, 28, 35
- in_dll () (*méthode ctypes._CData*), 709
- in_table_a1 () (*dans le module stringprep*), 138
- in_table_b1 () (*dans le module stringprep*), 138
- in_table_c3 () (*dans le module stringprep*), 138
- in_table_c4 () (*dans le module stringprep*), 139
- in_table_c5 () (*dans le module stringprep*), 139
- in_table_c6 () (*dans le module stringprep*), 139
- in_table_c7 () (*dans le module stringprep*), 139
- in_table_c8 () (*dans le module stringprep*), 139
- in_table_c9 () (*dans le module stringprep*), 139
- in_table_c11 () (*dans le module stringprep*), 138
- in_table_c11_c12 () (*dans le module stringprep*), 138
- in_table_c12 () (*dans le module stringprep*), 138
- in_table_c21 () (*dans le module stringprep*), 138
- in_table_c21_c22 () (*dans le module stringprep*), 138
- in_table_c22 () (*dans le module stringprep*), 138
- in_table_d1 () (*dans le module stringprep*), 139
- in_table_d2 () (*dans le module stringprep*), 139
- in_transaction (*attribut sqlite3.Connection*), 422
- inch () (*méthode curses.window*), 660
- inclusive (*attribut tracemalloc.DomainFilter*), 1516
- inclusive (*attribut tracemalloc.Filter*), 1516
- Incomplete, 1020
- IncompleteRead, 1133
- IncompleteReadError, 895
- increment_lineno () (*dans le module ast*), 1647
- incrementaldecoder (*attribut codecs.CodecInfo*), 151
- IncrementalDecoder (*classe dans codecs*), 157
- incrementalencoder (*attribut codecs.CodecInfo*), 151
- IncrementalEncoder (*classe dans codecs*), 156
- IncrementalNewlineDecoder (*classe dans io*), 572
- IncrementalParser (*classe dans xml.sax.xmlreader*), 1070
- indent (*attribut doctest.Example*), 1379
- INDENT (*dans le module token*), 1651
- indent () (*dans le module textwrap*), 133
- IndentationError, 85
- indentlevel=<num>
 - pickletools command line option, 1675
- index () (*dans le module operator*), 343
- index () (*méthode array.array*), 230
- index () (*méthode bytearray*), 54
- index () (*méthode bytes*), 54
- index () (*méthode collections.deque*), 208
- index () (*méthode str*), 44
- index () (*méthode tkinter.ttk.Notebook*), 1320
- index () (*méthode tkinter.ttk.Treeview*), 1326
- index () (*sequence method*), 35
- IndexError, 83
- indexOf () (*dans le module operator*), 345
- IndexSizeErr, 1054
- indication de type, 1764
- inet_aton () (*dans le module socket*), 805
- inet_ntoa () (*dans le module socket*), 805
- inet_ntop () (*dans le module socket*), 805
- inet_pton () (*dans le module socket*), 805
- Inexact (*classe dans decimal*), 298
- inf (*dans le module cmath*), 280
- inf (*dans le module math*), 277
- infile
 - json.tool command line option, 994
- infile (*attribut shlex.shlex*), 1300
- Infinity, 11
- infj (*dans le module cmath*), 280
- info
 - zipapp command line option, 1532
- info () (*dans le module logging*), 625
- info () (*méthode dis.Bytecode*), 1663
- info () (*méthode gettext.NullTranslations*), 1246
- info () (*méthode logging.Logger*), 617
- infolist () (*méthode zipfile.ZipFile*), 455
- .ini
 - file, 478
- ini file, 478
- init () (*dans le module mimetypes*), 1013
- init_color () (*dans le module curses*), 653
- init_database () (*dans le module msilib*), 1684
- init_pair () (*dans le module curses*), 653
- inited (*dans le module mimetypes*), 1013
- initgroups () (*dans le module os*), 523
- initial_indent (*attribut textwrap.TextWrapper*), 134
- initscr () (*dans le module curses*), 653
- inode () (*méthode os.DirEntry*), 541
- INPLACE_ADD (*opcode*), 1667
- INPLACE_AND (*opcode*), 1668
- INPLACE_FLOOR_DIVIDE (*opcode*), 1667
- INPLACE_LSHIFT (*opcode*), 1667
- INPLACE_MATRIX_MULTIPLY (*opcode*), 1667

- INPLACE_MODULO (*opcode*), 1667
- INPLACE_MULTIPLY (*opcode*), 1667
- INPLACE_OR (*opcode*), 1668
- INPLACE_POWER (*opcode*), 1667
- INPLACE_RSHIFT (*opcode*), 1667
- INPLACE_SUBTRACT (*opcode*), 1667
- INPLACE_TRUE_DIVIDE (*opcode*), 1667
- INPLACE_XOR (*opcode*), 1668
- input (2to3 *fixer*), 1471
- input () (*dans le module fileinput*), 372
- input () (*fonction de base*), 13
- input_charset (*attribut email.charset.Charset*), 979
- input_codec (*attribut email.charset.Charset*), 979
- InputOnly (*classe dans tkinter.tix*), 1334
- InputSource (*classe dans xml.sax.xmlreader*), 1070
- insch () (*méthode curses.window*), 660
- insdelln () (*méthode curses.window*), 660
- insert () (*méthode array.array*), 230
- insert () (*méthode collections.deque*), 208
- insert () (*méthode tkinter.ttk.Notebook*), 1320
- insert () (*méthode tkinter.ttk.Treeview*), 1326
- insert () (*méthode xml.etree.ElementTree.Element*), 1040
- insert () (*sequence method*), 37
- insert_text () (*dans le module readline*), 140
- insertBefore () (*méthode xml.dom.Node*), 1049
- insertln () (*méthode curses.window*), 660
- insnstr () (*méthode curses.window*), 660
- insort () (*dans le module bisect*), 227
- insort_left () (*dans le module bisect*), 227
- insort_right () (*dans le module bisect*), 227
- inspect (*module*), 1591
- inspect command line option
 - details, 1605
- InspectLoader (*classe dans importlib.abc*), 1627
- insstr () (*méthode curses.window*), 660
- install () (*dans le module gettext*), 1245
- install () (*méthode gettext.NullTranslations*), 1246
- install_opener () (*dans le module urllib.request*), 1105
- install_scripts () (*méthode venv.EnvBuilder*), 1527
- installHandler () (*dans le module unittest*), 1413
- instate () (*méthode tkinter.ttk.Widget*), 1318
- instr () (*méthode curses.window*), 660
- istream (*attribut shlex.shlex*), 1301
- instruction, 1763
- Instruction (*classe dans dis*), 1665
- Instruction.arg (*dans le module dis*), 1665
- Instruction.argrepr (*dans le module dis*), 1665
- Instruction.argval (*dans le module dis*), 1665
- Instruction.is_jump_target (*dans le module dis*), 1665
- Instruction.offset (*dans le module dis*), 1665
- Instruction.opcode (*dans le module dis*), 1665
- Instruction.opname (*dans le module dis*), 1665
- Instruction.starts_line (*dans le module dis*), 1665
- int
 - fonction de base, 29
- int (*attribut uuid.UUID*), 1171
- int (*classe de base*), 13
- Int2AP () (*dans le module imaplib*), 1147
- int_info (*dans le module sys*), 1547
- integer
 - literals, 29
 - objet, 29
 - types, operations on, 30
- Integral (*classe dans numbers*), 270
- Integrated Development Environment, 1336
- IntegrityError, 431
- Intel/DVI ADPCM, 1223
- IntEnum (*classe dans enum*), 251
- interact (*pdb command*), 1494
- interact () (*dans le module code*), 1611
- interact () (*méthode code.InteractiveConsole*), 1613
- interact () (*méthode telnetlib.Telnet*), 1169
- interactif, 1758
- InteractiveConsole (*classe dans code*), 1611
- InteractiveInterpreter (*classe dans code*), 1611
- intern (2to3 *fixer*), 1471
- intern () (*dans le module sys*), 1547
- internal_attr (*attribut zipfile.ZipInfo*), 459
- Internaldate2tuple () (*dans le module imaplib*), 1147
- internalSubset (*attribut xml.dom.DocumentType*), 1050
- Internet, 1085
- interpolation
 - bytearray (%), 62
 - bytes (%), 62
- interpolation, string (%), 49
- InterpolationDepthError, 494
- InterpolationError, 494
- InterpolationMissingOptionError, 494
- InterpolationSyntaxError, 494
- interprété, 1758
- interpreter prompts, 1549
- interrupt () (*méthode sqlite3.Connection*), 424
- interrupt_main () (*dans le module _thread*), 794
- InterruptedError, 88
- intersection () (*méthode frozenset*), 71
- intersection_update () (*méthode frozenset*), 72
- IntFlag (*classe dans enum*), 251
- intro (*attribut cmd.Cmd*), 1295
- InuseAttributeErr, 1054
- inv () (*dans le module operator*), 343
- InvalidAccessErr, 1054

- `invalidate_caches()` (dans le module `importlib`), 1624
- `invalidate_caches()` (méthode de la classe `importlib.machinery.PathFinder`), 1631
- `invalidate_caches()` (méthode `importlib.abc.MetaPathFinder`), 1626
- `invalidate_caches()` (méthode `importlib.abc.PathEntryFinder`), 1626
- `invalidate_caches()` (méthode `importlib.machinery.FileFinder`), 1631
- `InvalidCharacterErr`, 1054
- `InvalidModificationErr`, 1054
- `InvalidOperation` (classe dans `decimal`), 298
- `InvalidStateErr`, 1054
- `InvalidStateError`, 876
- `InvalidURL`, 1133
- `invert()` (dans le module `operator`), 343
- `IO` (classe dans `typing`), 1358
- `io` (module), 562
- `IOBase` (classe dans `io`), 565
- `ioctl()` (dans le module `fcntl`), 1710
- `ioctl()` (méthode `socket.socket`), 808
- `IOError`, 87
- `ior()` (dans le module `operator`), 348
- `io.StringIO`
 - objet, 41
- `ip` (attribut `ipaddress.IPv4Interface`), 1220
- `ip` (attribut `ipaddress.IPv6Interface`), 1220
- `ip_address()` (dans le module `ipaddress`), 1211
- `ip_interface()` (dans le module `ipaddress`), 1211
- `ip_network()` (dans le module `ipaddress`), 1211
- `ipaddress` (module), 1211
- `ipow()` (dans le module `operator`), 348
- `ipv4_mapped` (attribut `ipaddress.IPv6Address`), 1213
- `IPv4Address` (classe dans `ipaddress`), 1212
- `IPv4Interface` (classe dans `ipaddress`), 1220
- `IPv4Network` (classe dans `ipaddress`), 1215
- `IPv6Address` (classe dans `ipaddress`), 1213
- `IPv6Interface` (classe dans `ipaddress`), 1220
- `IPv6Network` (classe dans `ipaddress`), 1217
- `irshift()` (dans le module `operator`), 348
- `is`
 - opérateur, 28
- `is not`
 - opérateur, 28
- `is_()` (dans le module `operator`), 343
- `is_absolute()` (méthode `pathlib.PurePath`), 358
- `is_alive()` (méthode `multiprocessing.Process`), 734
- `is_alive()` (méthode `threading.Thread`), 718
- `is_assigned()` (méthode `symtable.Symbol`), 1650
- `is_attachment()` (méthode `email.message.EmailMessage`), 936
- `is_authenticated()` (méthode `url-lib.request.HTTPPasswordMgrWithPriorAuth`), 1114
- `is_block_device()` (méthode `pathlib.Path`), 364
- `is_blocked()` (méthode `http.cookiejar.DefaultCookiePolicy`), 1195
- `is_canonical()` (méthode `decimal.Context`), 295
- `is_canonical()` (méthode `decimal.Decimal`), 289
- `is_char_device()` (méthode `pathlib.Path`), 364
- `IS_CHARACTER_JUNK()` (dans le module `difflib`), 126
- `is_check_supported()` (dans le module `lzma`), 451
- `is_closed()` (méthode `asyncio.AbstractEventLoop`), 858
- `is_closing()` (méthode `asyncio.BaseTransport`), 883
- `is_declared_global()` (méthode `symtable.Symbol`), 1650
- `is_dir()` (méthode `os.DirEntry`), 541
- `is_dir()` (méthode `pathlib.Path`), 363
- `is_dir()` (méthode `zipfile.ZipInfo`), 459
- `is_enabled()` (dans le module `faulthandler`), 1488
- `is_expired()` (méthode `http.cookiejar.Cookie`), 1197
- `is_fifo()` (méthode `pathlib.Path`), 363
- `is_file()` (méthode `os.DirEntry`), 542
- `is_file()` (méthode `pathlib.Path`), 363
- `is_finalizing()` (dans le module `sys`), 1547
- `is_finite()` (méthode `decimal.Context`), 295
- `is_finite()` (méthode `decimal.Decimal`), 289
- `is_free()` (méthode `symtable.Symbol`), 1650
- `is_global` (attribut `ipaddress.IPv4Address`), 1213
- `is_global` (attribut `ipaddress.IPv6Address`), 1213
- `is_global()` (méthode `symtable.Symbol`), 1650
- `is_hop_by_hop()` (dans le module `wsgiref.util`), 1097
- `is_imported()` (méthode `symtable.Symbol`), 1650
- `is_infinite()` (méthode `decimal.Context`), 295
- `is_infinite()` (méthode `decimal.Decimal`), 289
- `is_integer()` (méthode `float`), 32
- `is_jython` (dans le module `test.support`), 1476
- `IS_LINE_JUNK()` (dans le module `difflib`), 126
- `is_linetouched()` (méthode `curses.window`), 660
- `is_link_local` (attribut `ipaddress.IPv4Address`), 1213
- `is_link_local` (attribut `ipaddress.IPv4Network`), 1216
- `is_link_local` (attribut `ipaddress.IPv6Address`), 1213
- `is_link_local` (attribut `ipaddress.IPv6Network`), 1218
- `is_local()` (méthode `symtable.Symbol`), 1650
- `is_loopback` (attribut `ipaddress.IPv4Address`), 1213
- `is_loopback` (attribut `ipaddress.IPv4Network`), 1216
- `is_loopback` (attribut `ipaddress.IPv6Address`), 1213
- `is_loopback` (attribut `ipaddress.IPv6Network`), 1218
- `is_multicast` (attribut `ipaddress.IPv4Address`), 1212
- `is_multicast` (attribut `ipaddress.IPv4Network`), 1216
- `is_multicast` (attribut `ipaddress.IPv6Address`), 1213
- `is_multicast` (attribut `ipaddress.IPv6Network`), 1218
- `is_multipart()` (méthode `email.message.EmailMessage`), 933
- `is_multipart()` (méthode `email.message.Message`), 968

- `is_namespace()` (méthode `symtable.Symbol`), 1650
- `is_nan()` (méthode `decimal.Context`), 295
- `is_nan()` (méthode `decimal.Decimal`), 289
- `is_nested()` (méthode `symtable.SymbolTable`), 1649
- `is_normal()` (méthode `decimal.Context`), 295
- `is_normal()` (méthode `decimal.Decimal`), 289
- `is_not()` (dans le module `operator`), 343
- `is_not_allowed()` (méthode `http.cookiejar.DefaultCookiePolicy`), 1195
- `is_optimized()` (méthode `symtable.SymbolTable`), 1649
- `is_package()` (méthode `importlib.abc.InspectLoader`), 1628
- `is_package()` (méthode `importlib.abc.SourceLoader`), 1629
- `is_package()` (méthode `importlib.machinery.ExtensionFileLoader`), 1633
- `is_package()` (méthode `importlib.machinery.SourceFileLoader`), 1632
- `is_package()` (méthode `importlib.machinery.SourcelessFileLoader`), 1632
- `is_package()` (méthode `zipimport.zipimporter`), 1616
- `is_parameter()` (méthode `symtable.Symbol`), 1650
- `is_private` (attribut `ipaddress.IPv4Address`), 1212
- `is_private` (attribut `ipaddress.IPv4Network`), 1216
- `is_private` (attribut `ipaddress.IPv6Address`), 1213
- `is_private` (attribut `ipaddress.IPv6Network`), 1218
- `is_python_build()` (dans le module `sysconfig`), 1556
- `is_qnan()` (méthode `decimal.Context`), 295
- `is_qnan()` (méthode `decimal.Decimal`), 289
- `is_referenced()` (méthode `symtable.Symbol`), 1650
- `is_reserved` (attribut `ipaddress.IPv4Address`), 1213
- `is_reserved` (attribut `ipaddress.IPv4Network`), 1216
- `is_reserved` (attribut `ipaddress.IPv6Address`), 1213
- `is_reserved` (attribut `ipaddress.IPv6Network`), 1218
- `is_reserved()` (méthode `pathlib.PurePath`), 359
- `is_resource_enabled()` (dans le module `test.support`), 1476
- `is_running()` (méthode `asyncio.AbstractEventLoop`), 858
- `is_set()` (méthode `asyncio.Event`), 905
- `is_set()` (méthode `threading.Event`), 724
- `is_signed()` (méthode `decimal.Context`), 295
- `is_signed()` (méthode `decimal.Decimal`), 289
- `is_site_local` (attribut `ipaddress.IPv6Address`), 1213
- `is_site_local` (attribut `ipaddress.IPv6Network`), 1218
- `is_snan()` (méthode `decimal.Context`), 295
- `is_snan()` (méthode `decimal.Decimal`), 289
- `is_socket()` (méthode `pathlib.Path`), 363
- `is_subnormal()` (méthode `decimal.Context`), 295
- `is_subnormal()` (méthode `decimal.Decimal`), 289
- `is_symlink()` (méthode `os.DirEntry`), 542
- `is_symlink()` (méthode `pathlib.Path`), 363
- `is_tarfile()` (dans le module `tarfile`), 462
- `is_term_resized()` (dans le module `curses`), 653
- `is_tracing()` (dans le module `tracemalloc`), 1515
- `is_tracked()` (dans le module `gc`), 1589
- `is_unspecified` (attribut `ipaddress.IPv4Address`), 1213
- `is_unspecified` (attribut `ipaddress.IPv4Network`), 1216
- `is_unspecified` (attribut `ipaddress.IPv6Address`), 1213
- `is_unspecified` (attribut `ipaddress.IPv6Network`), 1218
- `is_wintouched()` (méthode `curses.window`), 661
- `is_zero()` (méthode `decimal.Context`), 295
- `is_zero()` (méthode `decimal.Decimal`), 289
- `is_zipfile()` (dans le module `zipfile`), 454
- `isabs()` (dans le module `os.path`), 369
- `isabstract()` (dans le module `inspect`), 1594
- `IsADirectoryError`, 88
- `isalnum()` (dans le module `curses.ascii`), 669
- `isalnum()` (méthode `bytearray`), 58
- `isalnum()` (méthode `bytes`), 58
- `isalnum()` (méthode `str`), 44
- `isalpha()` (dans le module `curses.ascii`), 669
- `isalpha()` (méthode `bytearray`), 59
- `isalpha()` (méthode `bytes`), 59
- `isalpha()` (méthode `str`), 44
- `isascii()` (dans le module `curses.ascii`), 670
- `isasynccgen()` (dans le module `inspect`), 1594
- `isasynccgenfunction()` (dans le module `inspect`), 1593
- `isatty()` (dans le module `os`), 528
- `isatty()` (méthode `chunk.Chunk`), 1235
- `isatty()` (méthode `io.IOBase`), 566
- `isawaitable()` (dans le module `inspect`), 1593
- `isblank()` (dans le module `curses.ascii`), 670
- `isblk()` (méthode `tarfile.TarInfo`), 467
- `isbuiltin()` (dans le module `inspect`), 1594
- `ischr()` (méthode `tarfile.TarInfo`), 467
- `isclass()` (dans le module `inspect`), 1593
- `isclose()` (dans le module `cmath`), 280
- `isclose()` (dans le module `math`), 273
- `iscntrl()` (dans le module `curses.ascii`), 670
- `iscode()` (dans le module `inspect`), 1594
- `iscoroutine()` (dans le module `asyncio`), 880
- `iscoroutine()` (dans le module `inspect`), 1593
- `iscoroutinefunction()` (dans le module `asyncio`), 880
- `iscoroutinefunction()` (dans le module `inspect`), 1593
- `isctrl()` (dans le module `curses.ascii`), 670
- `isDaemon()` (méthode `threading.Thread`), 719
- `isdatadescriptor()` (dans le module `inspect`), 1594
- `isdecimal()` (méthode `str`), 44
- `isdev()` (méthode `tarfile.TarInfo`), 467

- `isdigit()` (dans le module `curses.ascii`), 670
- `isdigit()` (méthode `bytearray`), 59
- `isdigit()` (méthode `bytes`), 59
- `isdigit()` (méthode `str`), 44
- `isdir()` (dans le module `os.path`), 369
- `isdir()` (méthode `tarfile.TarInfo`), 467
- `isdisjoint()` (méthode `frozenset`), 71
- `isdown()` (dans le module `turtle`), 1273
- `iselement()` (dans le module `xml.etree.ElementTree`), 1038
- `isEnabled()` (dans le module `gc`), 1588
- `isEnabledFor()` (méthode `logging.Logger`), 616
- `isendwin()` (dans le module `curses`), 653
- `ISEOF()` (dans le module `token`), 1651
- `isexpr()` (dans le module `parser`), 1641
- `isexpr()` (méthode `parser.ST`), 1642
- `isfifo()` (méthode `tarfile.TarInfo`), 467
- `isfile()` (dans le module `os.path`), 369
- `isfile()` (méthode `tarfile.TarInfo`), 466
- `isfinite()` (dans le module `cmath`), 280
- `isfinite()` (dans le module `math`), 274
- `isfirstline()` (dans le module `fileinput`), 373
- `isframe()` (dans le module `inspect`), 1594
- `isfunction()` (dans le module `inspect`), 1593
- `isgenerator()` (dans le module `inspect`), 1593
- `isgeneratorfunction()` (dans le module `inspect`), 1593
- `isgetsetdescriptor()` (dans le module `inspect`), 1594
- `isgraph()` (dans le module `curses.ascii`), 670
- `isidentifier()` (méthode `str`), 44
- `isinf()` (dans le module `cmath`), 280
- `isinf()` (dans le module `math`), 274
- `isinstance(2to3 fixer)`, 1471
- `isinstance()` (fonction de base), 13
- `iskeyword()` (dans le module `keyword`), 1653
- `isleap()` (dans le module `calendar`), 201
- `islice()` (dans le module `itertools`), 328
- `islink()` (dans le module `os.path`), 369
- `islnk()` (méthode `tarfile.TarInfo`), 467
- `islower()` (dans le module `curses.ascii`), 670
- `islower()` (méthode `bytearray`), 59
- `islower()` (méthode `bytes`), 59
- `islower()` (méthode `str`), 44
- `ismemberdescriptor()` (dans le module `inspect`), 1594
- `ismeta()` (dans le module `curses.ascii`), 670
- `ismethod()` (dans le module `inspect`), 1593
- `ismethoddescriptor()` (dans le module `inspect`), 1594
- `ismodule()` (dans le module `inspect`), 1593
- `ismount()` (dans le module `os.path`), 370
- `isnan()` (dans le module `cmath`), 280
- `isnan()` (dans le module `math`), 274
- `ISNONTERMINAL()` (dans le module `token`), 1651
- `isnumeric()` (méthode `str`), 44
- `isocalendar()` (méthode `datetime.date`), 175
- `isocalendar()` (méthode `datetime.datetime`), 182
- `isoformat()` (méthode `datetime.date`), 175
- `isoformat()` (méthode `datetime.datetime`), 182
- `isoformat()` (méthode `datetime.time`), 186
- `isolation_level` (attribut `sqlite3.Connection`), 422
- `isowekday()` (méthode `datetime.date`), 175
- `isowekday()` (méthode `datetime.datetime`), 182
- `isprint()` (dans le module `curses.ascii`), 670
- `isprintable()` (méthode `str`), 44
- `ispunct()` (dans le module `curses.ascii`), 670
- `isreadable()` (dans le module `pprint`), 245
- `isreadable()` (méthode `pprint.PrettyPrinter`), 245
- `isrecursive()` (dans le module `pprint`), 245
- `isrecursive()` (méthode `pprint.PrettyPrinter`), 245
- `isreg()` (méthode `tarfile.TarInfo`), 467
- `isReservedKey()` (méthode `http.cookies.Morsel`), 1188
- `isroutine()` (dans le module `inspect`), 1594
- `isSameNode()` (méthode `xml.dom.Node`), 1049
- `isspace()` (dans le module `curses.ascii`), 670
- `isspace()` (méthode `bytearray`), 59
- `isspace()` (méthode `bytes`), 59
- `isspace()` (méthode `str`), 45
- `isstdin()` (dans le module `fileinput`), 373
- `issubclass()` (fonction de base), 14
- `issubset()` (méthode `frozenset`), 71
- `issuite()` (dans le module `parser`), 1641
- `issuite()` (méthode `parser.ST`), 1642
- `issuperset()` (méthode `frozenset`), 71
- `issym()` (méthode `tarfile.TarInfo`), 467
- `ISTERMINAL()` (dans le module `token`), 1651
- `istitle()` (méthode `bytearray`), 59
- `istitle()` (méthode `bytes`), 59
- `istitle()` (méthode `str`), 45
- `itraceback()` (dans le module `inspect`), 1594
- `isub()` (dans le module `operator`), 348
- `isupper()` (dans le module `curses.ascii`), 670
- `isupper()` (méthode `bytearray`), 60
- `isupper()` (méthode `bytes`), 60
- `isupper()` (méthode `str`), 45
- `isvisible()` (dans le module `turtle`), 1276
- `isxdigit()` (dans le module `curses.ascii`), 670
- `item()` (méthode `tkinter.ttk.Treeview`), 1326
- `item()` (méthode `xml.dom.NamedNodeMap`), 1052
- `item()` (méthode `xml.dom.NodeList`), 1049
- `itemgetter()` (dans le module `operator`), 345
- `items()` (méthode `configparser.ConfigParser`), 492
- `items()` (méthode `dict`), 74
- `items()` (méthode `email.message.EmailMessage`), 935
- `items()` (méthode `email.message.Message`), 970
- `items()` (méthode `mailbox.Mailbox`), 997

items() (méthode *types.MappingProxyType*), 241
 items() (méthode *xml.etree.ElementTree.Element*), 1040
 itemsize (attribut *array.array*), 229
 itemsize (attribut *memoryview*), 70
 ItemsView (classe dans *collections.abc*), 220
 ItemsView (classe dans *typing*), 1356
 iter() (fonction de base), 14
 iter() (méthode *xml.etree.ElementTree.Element*), 1040
 iter() (méthode *xml.etree.ElementTree.ElementTree*), 1041
 iter_attachments() (méthode *email.message.EmailMessage*), 938
 iter_child_nodes() (dans le module *ast*), 1647
 iter_fields() (dans le module *ast*), 1647
 iter_importers() (dans le module *pkgutil*), 1618
 iter_modules() (dans le module *pkgutil*), 1618
 iter_parts() (méthode *email.message.EmailMessage*), 938
 iter_unpack() (dans le module *struct*), 146
 iter_unpack() (méthode *struct.Struct*), 150
 itérable, 1758
 Iterable (classe dans *collections.abc*), 219
 Iterable (classe dans *typing*), 1355
 itérable asynchrone, 1754
 itérateur, 1759
 itérateur asynchrone, 1754
 itérateur de générateur, 1757
 itérateur de générateur asynchrone, 1754
 Iterator (classe dans *collections.abc*), 220
 Iterator (classe dans *typing*), 1355
 iterator protocol, 34
 iterdecode() (dans le module *codecs*), 153
 iterdir() (méthode *pathlib.Path*), 364
 iterdump() (méthode *sqlite3.Connection*), 427
 iterencode() (dans le module *codecs*), 153
 iterencode() (méthode *json.JSONEncoder*), 991
 iterfind() (méthode *xml.etree.ElementTree.Element*), 1040
 iterfind() (méthode *xml.etree.ElementTree.ElementTree*), 1041
 iteritems() (méthode *mailbox.Mailbox*), 997
 iterkeys() (méthode *mailbox.Mailbox*), 997
 itermonthdates() (méthode *calendar.Calendar*), 199
 itermonthdays() (méthode *calendar.Calendar*), 199
 itermonthdays2() (méthode *calendar.Calendar*), 199
 iterparse() (dans le module *xml.etree.ElementTree*), 1038
 intertext() (méthode *xml.etree.ElementTree.Element*), 1040
 itertools (2to3 *fixer*), 1471
 itertools (module), 321
 itertools_imports (2to3 *fixer*), 1471

intervalues() (méthode *mailbox.Mailbox*), 997
 iterweekdays() (méthode *calendar.Calendar*), 199
 ITIMER_PROF (dans le module *signal*), 923
 ITIMER_REAL (dans le module *signal*), 923
 ITIMER_VIRTUAL (dans le module *signal*), 923
 ItimerError, 923
 itruediv() (dans le module *operator*), 348
 ixor() (dans le module *operator*), 348

J

-j N
 compileall command line option, 1660
 Jansen, Jack, 1021
 java_ver() (dans le module *platform*), 674
 join() (dans le module *os.path*), 370
 join() (méthode *asyncio.Queue*), 908
 join() (méthode *bytearray*), 54
 join() (méthode *bytes*), 54
 join() (méthode *multiprocessing.JoinableQueue*), 738
 join() (méthode *multiprocessing.pool.Pool*), 754
 join() (méthode *multiprocessing.Process*), 734
 join() (méthode *queue.Queue*), 792
 join() (méthode *str*), 45
 join() (méthode *threading.Thread*), 718
 join_thread() (méthode *multiprocessing.Queue*), 737
 JoinableQueue (classe dans *multiprocessing*), 738
 joinpath() (méthode *pathlib.PurePath*), 359
 js_output() (méthode *http.cookies.BaseCookie*), 1187
 js_output() (méthode *http.cookies.Morsel*), 1188
 json (module), 985
 JSONDecodeError, 991
 JSONDecoder (classe dans *json*), 989
 JSONEncoder (classe dans *json*), 990
 json.tool (module), 993
 json.tool command line option
 -h, 994
 --help, 994
 infile, 994
 outfile, 994
 --sort-keys, 994
 jump (*pdb* command), 1493
 JUMP_ABSOLUTE (opcode), 1672
 JUMP_FORWARD (opcode), 1671
 JUMP_IF_FALSE_OR_POP (opcode), 1672
 JUMP_IF_TRUE_OR_POP (opcode), 1672

K

kbhit() (dans le module *msvcrt*), 1690
 KDEDIR, 1087
 kevent() (dans le module *select*), 848
 key (attribut *http.cookies.Morsel*), 1188
 KEY_ALL_ACCESS (dans le module *winreg*), 1696
 KEY_CREATE_LINK (dans le module *winreg*), 1696
 KEY_CREATE_SUB_KEY (dans le module *winreg*), 1696

- KEY_ENUMERATE_SUB_KEYS (dans le module winreg), 1696
- KEY_EXECUTE (dans le module winreg), 1696
- KEY_NOTIFY (dans le module winreg), 1696
- KEY_QUERY_VALUE (dans le module winreg), 1696
- KEY_READ (dans le module winreg), 1696
- KEY_SET_VALUE (dans le module winreg), 1696
- KEY_WOW64_32KEY (dans le module winreg), 1696
- KEY_WOW64_64KEY (dans le module winreg), 1696
- KEY_WRITE (dans le module winreg), 1696
- KeyboardInterrupt, 83
- KeyError, 83
- keyname() (dans le module curses), 653
- keypad() (méthode curses.window), 661
- keyrefs() (méthode weakref.WeakKeyDictionary), 233
- keys() (méthode dict), 74
- keys() (méthode email.message.EmailMessage), 934
- keys() (méthode email.message.Message), 970
- keys() (méthode mailbox.Mailbox), 997
- keys() (méthode sqlite3.Row), 430
- keys() (méthode types.MappingProxyType), 241
- keys() (méthode xml.etree.ElementTree.Element), 1040
- KeysView (classe dans collections.abc), 220
- KeysView (classe dans typing), 1356
- keyword (module), 1653
- keywords (attribut functools.partial), 342
- kill() (dans le module os), 553
- kill() (méthode asyncio.asyncio.subprocess.Process), 901
- kill() (méthode asyncio.BaseSubprocessTransport), 885
- kill() (méthode subprocess.Popen), 781
- killchar() (dans le module curses), 653
- killpg() (dans le module os), 553
- kind (attribut inspect.Parameter), 1597
- knownfiles (dans le module mimetypes), 1013
- kqueue() (dans le module select), 848
- KqueueSelector (classe dans selectors), 856
- kwargs (attribut inspect.BindArguments), 1598
- kwlist (dans le module keyword), 1653
- L**
- l
 - compileall command line option, 1660
 - pickletools command line option, 1675
 - trace command line option, 1508
- L (dans le module re), 109
- l <tarfile>
 - tarfile command line option, 468
- l <zipfile>
 - zipfile command line option, 460
- LabelEntry (classe dans tkinter.tix), 1332
- LabelFrame (classe dans tkinter.tix), 1332
- lambda, 1759
- LambdaType (dans le module types), 240
- LANG, 1244, 1245, 1252, 1255
- LANGUAGE, 1244, 1245
- language
 - C, 29, 30
- large files, 1701
- LargeZipFile, 453
- last() (méthode nntplib.NNTP), 1157
- last_accepted (attribut multiprocessing.connection.Listener), 756
- last_traceback (dans le module sys), 1547
- last_type (dans le module sys), 1547
- last_value (dans le module sys), 1547
- lastChild (attribut xml.dom.Node), 1048
- lastcmd (attribut cmd.Cmd), 1295
- lastgroup (attribut re.match), 116
- lastindex (attribut re.match), 116
- lastResort (dans le module logging), 628
- lastrowid (attribut sqlite3.Cursor), 429
- layout() (méthode tkinter.ttk.Style), 1328
- lazycache() (dans le module linecache), 388
- LazyLoader (classe dans importlib.util), 1636
- LBACE (dans le module token), 1651
- LBYL, 1759
- LC_ALL, 1244, 1245
- LC_ALL (dans le module locale), 1257
- LC_COLLATE (dans le module locale), 1257
- LC_CTYPE (dans le module locale), 1257
- LC_MESSAGES, 1244, 1245
- LC_MESSAGES (dans le module locale), 1257
- LC_MONETARY (dans le module locale), 1257
- LC_NUMERIC (dans le module locale), 1257
- LC_TIME (dans le module locale), 1257
- lchflags() (dans le module os), 536
- lchmod() (dans le module os), 536
- lchmod() (méthode pathlib.Path), 364
- lchown() (dans le module os), 536
- ldexp() (dans le module math), 274
- ldgettext() (dans le module gettext), 1244
- ldngettext() (dans le module gettext), 1244
- Le zen de Python, 1764
- le() (dans le module operator), 342
- leapdays() (dans le module calendar), 201
- leaveok() (méthode curses.window), 661
- left (attribut filecmp.dircmp), 380
- left() (dans le module turtle), 1266
- left_list (attribut filecmp.dircmp), 380
- left_only (attribut filecmp.dircmp), 380
- LEFTSHIFT (dans le module token), 1651
- LEFTSHIFTEQUAL (dans le module token), 1651
- len
 - fonction de base, 35, 73
- len() (fonction de base), 14
- length (attribut xml.dom.NamedNodeMap), 1052

- `length` (attribut `xml.dom.NodeList`), 1049
- `length_hint()` (dans le module `operator`), 345
- `LESS` (dans le module `token`), 1651
- `LESSEQUAL` (dans le module `token`), 1651
- `lexists()` (dans le module `os.path`), 368
- `lgamma()` (dans le module `math`), 277
- `gettext()` (dans le module `gettext`), 1244
- `gettext()` (méthode `gettext.GNUTranslations`), 1247
- `gettext()` (méthode `gettext.NullTranslations`), 1246
- `lib2to3` (module), 1473
- `libc_ver()` (dans le module `platform`), 675
- `library` (attribut `ssl.SSLError`), 818
- `library` (dans le module `dbm.ndbm`), 417
- `LibraryLoader` (classe dans `ctypes`), 703
- `license` (variable de base), 26
- `LifoQueue` (classe dans `asyncio`), 908
- `LifoQueue` (classe dans `queue`), 791
- light-weight processes, 794
- `limit_denominator()` (méthode `fractions.Fraction`), 307
- `LimitOverrunError`, 895
- `lin2adpcm()` (dans le module `audioop`), 1224
- `lin2alaw()` (dans le module `audioop`), 1224
- `lin2lin()` (dans le module `audioop`), 1224
- `lin2ulaw()` (dans le module `audioop`), 1225
- `line()` (méthode `msilib.Dialog`), 1688
- `line_buffering` (attribut `io.TextIOWrapper`), 572
- `line_num` (attribut `csv.csvreader`), 476
- line-buffered I/O, 17
- `linecache` (module), 387
- `lineno` (attribut `ast.AST`), 1644
- `lineno` (attribut `doctest.DocTest`), 1379
- `lineno` (attribut `doctest.Example`), 1379
- `lineno` (attribut `json.JSONDecodeError`), 991
- `lineno` (attribut `pyclbr.Class`), 1658
- `lineno` (attribut `pyclbr.Function`), 1658
- `lineno` (attribut `re.error`), 113
- `lineno` (attribut `shlex.shlex`), 1301
- `lineno` (attribut `traceback.TracebackException`), 1583
- `lineno` (attribut `tracemalloc.Filter`), 1516
- `lineno` (attribut `tracemalloc.Frame`), 1517
- `lineno` (attribut `xml.parsers.expat.ExpatError`), 1079
- `lineno()` (dans le module `fileinput`), 372
- `LINES`, 652, 656, 657
- `lines` (attribut `os.terminal_size`), 532
- `linesep` (attribut `email.policy.Policy`), 947
- `linesep` (dans le module `os`), 561
- `lineterminator` (attribut `csv.Dialect`), 475
- `LineTooLong`, 1134
- `link()` (dans le module `os`), 536
- `linkname` (attribut `tarfile.TarInfo`), 466
- `linux_distribution()` (dans le module `platform`), 675
- `list`, 1759
 - objet, 37, 38
 - type, operations on, 37
- `List` (classe dans `typing`), 1356
- `list` (classe de base), 38
- `list` (*pdb* command), 1494
- `--list <tarfile>`
 - `tarfile` command line option, 468
- `list()` (méthode `imaplib.IMAP4`), 1149
- `list()` (méthode `multiprocessing.managers.SyncManager`), 748
- `list()` (méthode `nntplib.NNTP`), 1155
- `list()` (méthode `poplib.POP3`), 1145
- `list()` (méthode `tarfile.TarFile`), 464
- `LIST_APPEND` (opcode), 1668
- `list_dialects()` (dans le module `csv`), 473
- `list_folders()` (méthode `mailbox.Maildir`), 999
- `list_folders()` (méthode `mailbox.MH`), 1001
- `listdir()` (dans le module `os`), 537
- liste en compréhension (ou liste en intension), 1759
- `listen()` (dans le module `logging.config`), 630
- `listen()` (dans le module `turtle`), 1283
- `listen()` (méthode `asyncore.dispatcher`), 917
- `listen()` (méthode `socket.socket`), 808
- `Listener` (classe dans `multiprocessing.connection`), 755
- `--listfuncs`
 - `trace` command line option, 1508
- `listMethods()` (méthode `xmlrpc.client.ServerProxy.system`), 1200
- `ListNoteBook` (classe dans `tkinter.tix`), 1333
- `listxattr()` (dans le module `os`), 549
- `literal_eval()` (dans le module `ast`), 1647
- literals
 - binary, 29
 - complex number, 29
 - floating point, 29
 - hexadecimal, 29
 - integer, 29
 - numeric, 29
 - octal, 29
- `LittleEndianStructure` (classe dans `ctypes`), 711
- `ljust()` (méthode `bytearray`), 56
- `ljust()` (méthode `bytes`), 56
- `ljust()` (méthode `str`), 45
- `LK_LOCK` (dans le module `msvcrt`), 1689
- `LK_NBLCK` (dans le module `msvcrt`), 1689
- `LK_NBRLCK` (dans le module `msvcrt`), 1689
- `LK_RLCK` (dans le module `msvcrt`), 1689
- `LK_UNLCK` (dans le module `msvcrt`), 1689
- `ll` (*pdb* command), 1494
- `LMTP` (classe dans `smtplib`), 1159
- `ln()` (méthode `decimal.Context`), 295
- `ln()` (méthode `decimal.Decimal`), 289
- `LNAME`, 650

- `lngettext()` (dans le module `gettext`), 1244
`lngettext()` (méthode `gettext.GNUTranslations`), 1247
`lngettext()` (méthode `gettext.NullTranslations`), 1246
`load()` (dans le module `json`), 988
`load()` (dans le module `marshal`), 414
`load()` (dans le module `pickle`), 399
`load()` (dans le module `plistlib`), 499
`load()` (méthode de la classe `tracemalloc.Snapshot`), 1517
`load()` (méthode `http.cookiejar.FileCookieJar`), 1192
`load()` (méthode `http.cookies.BaseCookie`), 1187
`load()` (méthode `pickle.Unpickler`), 401
`LOAD_ATTR` (opcode), 1671
`LOAD_BUILD_CLASS` (opcode), 1669
`load_cert_chain()` (méthode `ssl.SSLContext`), 832
`LOAD_CLASSDEREF` (opcode), 1672
`LOAD_CLOSURE` (opcode), 1672
`LOAD_CONST` (opcode), 1670
`load_default_certs()` (méthode `ssl.SSLContext`), 832
`LOAD_DEREF` (opcode), 1672
`load_dh_params()` (méthode `ssl.SSLContext`), 835
`load_extension()` (méthode `sqlite3.Connection`), 425
`LOAD_FAST` (opcode), 1672
`LOAD_GLOBAL` (opcode), 1672
`load_module()` (dans le module `imp`), 1745
`load_module()` (méthode `importlib.abc.FileLoader`), 1628
`load_module()` (méthode `importlib.abc.InspectLoader`), 1628
`load_module()` (méthode `importlib.abc.Loader`), 1627
`load_module()` (méthode `importlib.abc.SourceLoader`), 1629
`load_module()` (méthode `importlib.machinery.SourceFileLoader`), 1632
`load_module()` (méthode `importlib.machinery.SourcelessFileLoader`), 1632
`load_module()` (méthode `zipimport.zipimporter`), 1616
`LOAD_NAME` (opcode), 1670
`load_package_tests()` (dans le module `test.support`), 1479
`load_verify_locations()` (méthode `ssl.SSLContext`), 833
`loader` (attribut `importlib.machinery.ModuleSpec`), 1633
`Loader` (classe dans `importlib.abc`), 1626
`loader_state` (attribut `importlib.machinery.ModuleSpec`), 1633
`LoadError`, 1190
`LoadKey()` (dans le module `winreg`), 1693
`LoadLibrary()` (méthode `ctypes.LibraryLoader`), 703
`loads()` (dans le module `json`), 989
`loads()` (dans le module `marshal`), 414
`loads()` (dans le module `pickle`), 400
`loads()` (dans le module `plistlib`), 499
`loads()` (dans le module `xmlrpc.client`), 1204
`loadTestsFromModule()` (méthode `unittest.TestLoader`), 1405
`loadTestsFromName()` (méthode `unittest.TestLoader`), 1406
`loadTestsFromNames()` (méthode `unittest.TestLoader`), 1406
`loadTestsFromTestCase()` (méthode `unittest.TestLoader`), 1405
`local` (classe dans `threading`), 717
`localcontext()` (dans le module `decimal`), 292
`LOCALE` (dans le module `re`), 109
`locale` (module), 1251
`localeconv()` (dans le module `locale`), 1252
`LocaleHTMLCalendar` (classe dans `calendar`), 200
`LocaleTextCalendar` (classe dans `calendar`), 200
`localName` (attribut `xml.dom.Attr`), 1052
`localName` (attribut `xml.dom.Node`), 1048
`--locals`
`unittest` command line option, 1389
`locals()` (fonction de base), 14
`localtime()` (dans le module `email.utils`), 982
`localtime()` (dans le module `time`), 575
`Locator` (classe dans `xml.sax.xmlreader`), 1070
`Lock` (classe dans `asyncio`), 904
`Lock` (classe dans `multiprocessing`), 742
`Lock` (classe dans `threading`), 719
`lock()` (méthode `mailbox.Babyl`), 1002
`lock()` (méthode `mailbox.Mailbox`), 998
`lock()` (méthode `mailbox.Maildir`), 999
`lock()` (méthode `mailbox.mbox`), 1000
`lock()` (méthode `mailbox.MH`), 1001
`lock()` (méthode `mailbox.MMDF`), 1003
`Lock()` (méthode `multiprocessing.managers.SyncManager`), 748
`lock_held()` (dans le module `imp`), 1747
`locked()` (méthode `_thread.lock`), 795
`locked()` (méthode `asyncio.Condition`), 906
`locked()` (méthode `asyncio.Lock`), 905
`locked()` (méthode `asyncio.Semaphore`), 907
`lockf()` (dans le module `fcntl`), 1710
`lockf()` (dans le module `os`), 528
`locking()` (dans le module `msvcrt`), 1689
`LockType` (dans le module `_thread`), 794
`log()` (dans le module `cmath`), 279
`log()` (dans le module `logging`), 626
`log()` (dans le module `math`), 275
`log()` (méthode `logging.Logger`), 617
`log1p()` (dans le module `math`), 275
`log2()` (dans le module `math`), 275
`log10()` (dans le module `cmath`), 279
`log10()` (dans le module `math`), 275
`log10()` (méthode `decimal.Context`), 295
`log10()` (méthode `decimal.Decimal`), 289

- `log_date_time_string()` (méthode `http.server.BaseHTTPRequestHandler`), 1184
 - `log_error()` (méthode `http.server.BaseHTTPRequestHandler`), 1184
 - `log_exception()` (méthode `wsgi-ref.handlers.BaseHandler`), 1102
 - `log_message()` (méthode `http.server.BaseHTTPRequestHandler`), 1184
 - `log_request()` (méthode `http.server.BaseHTTPRequestHandler`), 1184
 - `log_to_stderr()` (dans le module `multiprocessing`), 758
 - `logb()` (méthode `decimal.Context`), 295
 - `logb()` (méthode `decimal.Decimal`), 289
 - `Logger` (classe dans `logging`), 615
 - `LoggerAdapter` (classe dans `logging`), 624
 - `logging`
 - `Errors`, 614
 - `logging` (module), 614
 - `logging.config` (module), 629
 - `logging.handlers` (module), 638
 - `logical_and()` (méthode `decimal.Context`), 295
 - `logical_and()` (méthode `decimal.Decimal`), 289
 - `logical_invert()` (méthode `decimal.Context`), 295
 - `logical_invert()` (méthode `decimal.Decimal`), 289
 - `logical_or()` (méthode `decimal.Context`), 295
 - `logical_or()` (méthode `decimal.Decimal`), 289
 - `logical_xor()` (méthode `decimal.Context`), 295
 - `logical_xor()` (méthode `decimal.Decimal`), 289
 - `login()` (méthode `ftplib.FTP`), 1141
 - `login()` (méthode `imaplib.IMAP4`), 1149
 - `login()` (méthode `nnplib.NNTP`), 1155
 - `login()` (méthode `smtplib.SMTP`), 1161
 - `login_cram_md5()` (méthode `imaplib.IMAP4`), 1149
 - `LOGNAME`, 522, 650
 - `lognormvariate()` (dans le module `random`), 311
 - `logout()` (méthode `imaplib.IMAP4`), 1149
 - `LogRecord` (classe dans `logging`), 621
 - `long` (2to3 fixer), 1471
 - `longMessage` (attribut `unittest.TestCase`), 1402
 - `longname()` (dans le module `curses`), 654
 - `lookup()` (dans le module `codecs`), 151
 - `lookup()` (dans le module `unicodedata`), 136
 - `lookup()` (méthode `symtable.SymbolTable`), 1649
 - `lookup()` (méthode `tkinter.ttk.Style`), 1328
 - `lookup_error()` (dans le module `codecs`), 155
 - `LookupError`, 82
 - `loop()` (dans le module `asyncore`), 915
 - `lower()` (méthode `bytearray`), 60
 - `lower()` (méthode `bytes`), 60
 - `lower()` (méthode `str`), 45
 - `LPAR` (dans le module `token`), 1651
 - `lru_cache()` (dans le module `functools`), 336
 - `lseek()` (dans le module `os`), 528
 - `lshift()` (dans le module `operator`), 343
 - `LSQB` (dans le module `token`), 1651
 - `lstat()` (dans le module `os`), 537
 - `lstat()` (méthode `pathlib.Path`), 364
 - `lstrip()` (méthode `bytearray`), 56
 - `lstrip()` (méthode `bytes`), 56
 - `lstrip()` (méthode `str`), 45
 - `lsub()` (méthode `imaplib.IMAP4`), 1149
 - `lt()` (dans le module `operator`), 342
 - `lt()` (dans le module `turtle`), 1266
 - `LWPCookieJar` (classe dans `http.cookiejar`), 1193
 - `lzma` (module), 448
 - `LZMACompressor` (classe dans `lzma`), 449
 - `LZMADecompressor` (classe dans `lzma`), 450
 - `LZMAError`, 448
 - `LZMAFile` (classe dans `lzma`), 448
- ## M
- `-m`
 - `pickletools` command line option, 1675
 - `trace` command line option, 1509
 - `M` (dans le module `re`), 109
 - `-m <mainfn>`
 - `zipapp` command line option, 1532
 - `mac_ver()` (dans le module `platform`), 675
 - `machine virtuelle`, 1764
 - `machine()` (dans le module `platform`), 672
 - `macpath` (module), 395
 - `macros` (attribut `netrc.netrc`), 495
 - `MAGIC_NUMBER` (dans le module `importlib.util`), 1634
 - `MagicMock` (classe dans `unittest.mock`), 1438
 - `Mailbox` (classe dans `mailbox`), 996
 - `mailbox` (module), 995
 - `mailcap` (module), 994
 - `Maildir` (classe dans `mailbox`), 998
 - `MaildirMessage` (classe dans `mailbox`), 1004
 - `mailfrom` (attribut `smtplib.SMTPChannel`), 1167
 - `MailmanProxy` (classe dans `smtplib`), 1166
 - `main()` (dans le module `py_compile`), 1659
 - `main()` (dans le module `site`), 1607
 - `main()` (dans le module `unittest`), 1409
 - `--main=<mainfn>`
 - `zipapp` command line option, 1532
 - `main_thread()` (dans le module `threading`), 716
 - `mainloop()` (dans le module `turtle`), 1284
 - `maintype` (attribut `email.headerregistry.ContentTypeHeader`), 956
 - `major` (attribut `email.headerregistry.MIMEVersionHeader`), 955
 - `major()` (dans le module `os`), 538
 - `make_alternative()` (méthode `email.message.EmailMessage`), 938
 - `make_archive()` (dans le module `shutil`), 393

- ul style="list-style-type: none; padding-left: 0;">
- `make_bad_fd()` (dans le module `test.support`), 1478
- `make_cookies()` (méthode `http.cookiejar.CookieJar`), 1192
- `make_file()` (méthode `difflib.HtmlDiff`), 123
- `MAKE_FUNCTION` (opcode), 1673
- `make_header()` (dans le module `email.header`), 978
- `make_mixed()` (méthode `email.message.EmailMessage`), 938
- `make_msgid()` (dans le module `email.utils`), 982
- `make_parser()` (dans le module `xml.sax`), 1062
- `make_related()` (méthode `email.message.EmailMessage`), 938
- `make_server()` (dans le module `wsgiref.simple_server`), 1099
- `make_table()` (méthode `difflib.HtmlDiff`), 123
- `makedev()` (dans le module `os`), 538
- `makedirs()` (dans le module `os`), 537
- `makeelement()` (méthode `xml.etree.ElementTree.Element`), 1041
- `makefile()` (méthode `socket.socket`), 808
- `makeLogRecord()` (dans le module `logging`), 626
- `makePickle()` (méthode `logging.handlers.SocketHandler`), 643
- `makeRecord()` (méthode `logging.Logger`), 617
- `makeSocket()` (méthode `logging.handlers.DatagramHandler`), 644
- `makeSocket()` (méthode `logging.handlers.SocketHandler`), 643
- `maketrans()` (méthode statique `bytearray`), 54
- `maketrans()` (méthode statique `bytes`), 54
- `maketrans()` (méthode statique `str`), 45
- `mangle_from_` (attribut `email.policy.Compat32`), 951
- `mangle_from_` (attribut `email.policy.Policy`), 948
- `map` (2to3 fixer), 1471
- `map()` (fonction de base), 14
- `map()` (méthode `concurrent.futures.Executor`), 768
- `map()` (méthode `multiprocessing.pool.Pool`), 753
- `map()` (méthode `tkinter.ttk.Style`), 1328
- `MAP_ADD` (opcode), 1668
- `map_async()` (méthode `multiprocessing.pool.Pool`), 753
- `map_table_b2()` (dans le module `stringprep`), 138
- `map_table_b3()` (dans le module `stringprep`), 138
- `map_to_type()` (méthode `email.headerregistry.HeaderRegistry`), 956
- `mapLogRecord()` (méthode `logging.handlers.HTTPHandler`), 648
- `mapping`
 - objet, 73
 - types, operations on, 73
- `Mapping` (classe dans `collections.abc`), 220
- `Mapping` (classe dans `typing`), 1355
- `mapping()` (méthode `msilib.Control`), 1687
- `MappingProxyType` (classe dans `types`), 241
- `MapView` (classe dans `collections.abc`), 220
- `MapView` (classe dans `typing`), 1356
- `mapPriority()` (méthode `logging.handlers.SysLogHandler`), 645
- `maps` (attribut `collections.ChainMap`), 202
- `maps()` (dans le module `nis`), 1716
- `marshal` (module), 413
- `marshalling`
 - objects, 397
- `masking`
 - operations, 30
- `Match` (classe dans `typing`), 1358
- `match()` (dans le module `nis`), 1716
- `match()` (dans le module `re`), 110
- `match()` (méthode `pathlib.PurePath`), 359
- `match()` (méthode `re.regex`), 113
- `match_hostname()` (dans le module `ssl`), 822
- `math`
 - module, 30, 281
- `math` (module), 272
- `matmul()` (dans le module `operator`), 344
- `max`
 - fonction de base, 35
- `max` (attribut `datetime.date`), 174
- `max` (attribut `datetime.datetime`), 178
- `max` (attribut `datetime.time`), 185
- `max` (attribut `datetime.timedelta`), 171
- `max()` (dans le module `audioop`), 1225
- `max()` (fonction de base), 14
- `max()` (méthode `decimal.Context`), 295
- `max()` (méthode `decimal.Decimal`), 289
- `max_count` (attribut `email.headerregistry.BaseHeader`), 954
- `MAX_EMAX` (dans le module `decimal`), 297
- `MAX_INTERPOLATION_DEPTH` (dans le module `configparser`), 493
- `max_line_length` (attribut `email.policy.Policy`), 947
- `max_lines` (attribut `textwrap.TextWrapper`), 135
- `max_mag()` (méthode `decimal.Context`), 295
- `max_mag()` (méthode `decimal.Decimal`), 290
- `MAX_PREC` (dans le module `decimal`), 297
- `max_prefixlen` (attribut `ipaddress.IPv4Address`), 1212
- `max_prefixlen` (attribut `ipaddress.IPv4Network`), 1216
- `max_prefixlen` (attribut `ipaddress.IPv6Address`), 1213
- `max_prefixlen` (attribut `ipaddress.IPv6Network`), 1218
- `maxarray` (attribut `reprlib.Repr`), 250
- `maxdeque` (attribut `reprlib.Repr`), 250
- `maxdict` (attribut `reprlib.Repr`), 250
- `maxDiff` (attribut `unittest.TestCase`), 1402
- `maxfrozenset` (attribut `reprlib.Repr`), 250
- `maxlen` (attribut `collections.deque`), 208
- `maxlevel` (attribut `reprlib.Repr`), 250
- `maxlist` (attribut `reprlib.Repr`), 250
- `maxlength` (attribut `reprlib.Repr`), 250
- `maxother` (attribut `reprlib.Repr`), 250

- `maxpp()` (dans le module `audioop`), 1225
- `maxset` (attribut `reprlib.Repr`), 250
- `maxsize` (attribut `asyncio.Queue`), 908
- `maxsize` (dans le module `sys`), 1547
- `maxstring` (attribut `reprlib.Repr`), 250
- `maxtuple` (attribut `reprlib.Repr`), 250
- `maxunicode` (dans le module `sys`), 1548
- `MAXYEAR` (dans le module `datetime`), 170
- `MB_ICONASTERISK` (dans le module `winsound`), 1699
- `MB_ICONEXCLAMATION` (dans le module `winsound`), 1699
- `MB_ICONHAND` (dans le module `winsound`), 1699
- `MB_ICONQUESTION` (dans le module `winsound`), 1699
- `MB_OK` (dans le module `winsound`), 1700
- `mbox` (classe dans `mailbox`), 1000
- `mboxMessage` (classe dans `mailbox`), 1005
- `mean()` (dans le module `statistics`), 316
- `median()` (dans le module `statistics`), 316
- `median_grouped()` (dans le module `statistics`), 317
- `median_high()` (dans le module `statistics`), 317
- `median_low()` (dans le module `statistics`), 317
- `MemberDescriptorType` (dans le module `types`), 241
- `memmove()` (dans le module `ctypes`), 707
- `--memo`
 - `pickletools` command line option, 1675
- `MemoryBIO` (classe dans `ssl`), 844
- `MemoryError`, 83
- `MemoryHandler` (classe dans `logging.handlers`), 647
- `memoryview`
 - objet, 51
- `memoryview` (classe de base), 64
- `memset()` (dans le module `ctypes`), 707
- `merge()` (dans le module `heapq`), 223
- `Message` (classe dans `email.message`), 967
- `Message` (classe dans `mailbox`), 1003
- `message digest`, MD5, 503
- `message_factory` (attribut `email.policy.Policy`), 948
- `message_from_binary_file()` (dans le module `email`), 942
- `message_from_bytes()` (dans le module `email`), 942
- `message_from_file()` (dans le module `email`), 942
- `message_from_string()` (dans le module `email`), 942
- `MessageBeep()` (dans le module `winsound`), 1698
- `MessageClass` (attribut `http.server.BaseHTTPRequestHandler`), 1183
- `MessageError`, 952
- `MessageParseError`, 952
- `messages` (dans le module `xml.parsers.expat.errors`), 1081
- `meta()` (dans le module `curses`), 654
- `meta_path` (dans le module `sys`), 1548
- `metaclass` (2to3 fixer), 1471
- `métaclasses`, 1760
- `MetaPathFinder` (classe dans `importlib.abc`), 1625
- `metavar` (attribut `optparse.Option`), 1732
- `MetavarTypeHelpFormatter` (classe dans `argparse`), 587
- `Meter` (classe dans `tkinter.tix`), 1332
- `method`
 - objet, 77
- `method` (attribut `urllib.request.Request`), 1109
- `method_calls` (attribut `unittest.mock.Mock`), 1422
- `METHOD_CRYPT` (dans le module `crypt`), 1705
- `METHOD_MD5` (dans le module `crypt`), 1705
- `METHOD_SHA256` (dans le module `crypt`), 1705
- `METHOD_SHA512` (dans le module `crypt`), 1705
- `methodattrs` (2to3 fixer), 1471
- `methodcaller()` (dans le module `operator`), 346
- `méthode`, 1760
- `méthode spéciale`, 1763
- `methodHelp()` (méthode `xmlrpc.client.ServerProxy.system`), 1200
- `methods`
 - `bytearray`, 53
 - `bytes`, 53
 - `string`, 42
- `methods` (attribut `pyclbr.Class`), 1658
- `methods` (dans le module `crypt`), 1705
- `methodSignature()` (méthode `xmlrpc.client.ServerProxy.system`), 1200
- `MethodType` (dans le module `types`), 240
- `MH` (classe dans `mailbox`), 1001
- `MHMessage` (classe dans `mailbox`), 1007
- `microsecond` (attribut `datetime.datetime`), 179
- `microsecond` (attribut `datetime.time`), 186
- `MIME`
 - `base64 encoding`, 1015
 - `content type`, 1012
 - `headers`, 1012, 1088
 - `quoted-printable encoding`, 1021
- `MIMEApplication` (classe dans `email.mime.application`), 975
- `MIMEAudio` (classe dans `email.mime.audio`), 975
- `MIMEBase` (classe dans `email.mime.base`), 974
- `MIMEImage` (classe dans `email.mime.image`), 976
- `MIMEMessage` (classe dans `email.mime.message`), 976
- `MIMEMultipart` (classe dans `email.mime.multipart`), 975
- `MIMENonMultipart` (classe dans `email.mime.nonmultipart`), 975
- `MIMEPart` (classe dans `email.message`), 939
- `MIMEText` (classe dans `email.mime.text`), 976
- `MimeTypes` (classe dans `mimetypes`), 1014
- `mimetypes` (module), 1012
- `MIMEVersionHeader` (classe dans `email.headerregistry`), 955

`min`
fonction de base, 35
`min` (attribut `datetime.date`), 174
`min` (attribut `datetime.datetime`), 178
`min` (attribut `datetime.time`), 185
`min` (attribut `datetime.timedelta`), 171
`min()` (fonction de base), 15
`min()` (méthode `decimal.Context`), 295
`min()` (méthode `decimal.Decimal`), 290
`MIN_EMIN` (dans le module `decimal`), 297
`MIN_ETINY` (dans le module `decimal`), 297
`min_mag()` (méthode `decimal.Context`), 295
`min_mag()` (méthode `decimal.Decimal`), 290
`MINEQUAL` (dans le module `token`), 1651
`minmax()` (dans le module `audioop`), 1225
`minor` (attribut `email.headerregistry.MIMEVersionHeader`), 956
`minor()` (dans le module `os`), 538
`MINUS` (dans le module `token`), 1651
`minus()` (méthode `decimal.Context`), 295
`minute` (attribut `datetime.datetime`), 179
`minute` (attribut `datetime.time`), 185
`MINYEAR` (dans le module `datetime`), 170
`mirrored()` (dans le module `unicodedata`), 136
`misc_header` (attribut `cmd.Cmd`), 1295
`--missing`
trace command line option, 1509
`MissingSectionHeaderError`, 494
`MIXERDEV`, 1238
`mkd()` (méthode `ftplib.FTP`), 1142
`mkdir()` (dans le module `os`), 537
`mkdir()` (méthode `pathlib.Path`), 364
`mkdtemp()` (dans le module `tempfile`), 382
`mkfifo()` (dans le module `os`), 538
`mknod()` (dans le module `os`), 538
`mksalt()` (dans le module `crypt`), 1706
`mkstemp()` (dans le module `tempfile`), 382
`mktemp()` (dans le module `tempfile`), 384
`mktime()` (dans le module `time`), 575
`mktime_tz()` (dans le module `email.utils`), 983
`mlsd()` (méthode `ftplib.FTP`), 1142
`mmap` (classe dans `mmap`), 926
`mmap` (module), 926
`M MDF` (classe dans `mailbox`), 1003
`M MDFMessage` (classe dans `mailbox`), 1009
`Mock` (classe dans `unittest.mock`), 1416
`mock_add_spec()` (méthode `unittest.mock.Mock`), 1419
`mock_calls` (attribut `unittest.mock.Mock`), 1422
`mock_open()` (dans le module `unittest.mock`), 1443
`mod()` (dans le module `operator`), 344
`mode` (attribut `io.FileIO`), 568
`mode` (attribut `ossaudiodev.oss_audio_device`), 1240
`mode` (attribut `tarfile.TarInfo`), 466
`mode()` (dans le module `statistics`), 318
`mode()` (dans le module `turtle`), 1285
`modes`
file, 16
`modf()` (dans le module `math`), 274
`modified()` (méthode `url-lib.robotparser.RobotFileParser`), 1129
`Modify()` (méthode `msilib.View`), 1685
`modify()` (méthode `select.devpoll`), 849
`modify()` (méthode `select.epoll`), 850
`modify()` (méthode `selectors.BaseSelector`), 855
`modify()` (méthode `select.poll`), 851
`module`, 1760
__main__, 1621, 1622
_locale, 1251
array, 51
base64, 1019
bdb, 1489
binhex, 1019
cmd, 1489
copy, 410
crypt, 1702
dbm.gnu, 412
dbm.ndbm, 412
errno, 84
glob, 386
imp, 23
math, 30, 281
os, 1701
pickle, 243, 410, 411, 413
pty, 529
pwd, 369
pyexpat, 1074
re, 42, 386
search path, 387, 1548, 1605
shelve, 413
signal, 795
sitecustomize, 1606
socket, 1085
stat, 542
string, 1257
struct, 812
sys, 17
types, 78
urllib.request, 1132
usercustomize, 1606
uu, 1019
`module` (attribut `pyclbr.Class`), 1658
`module` (attribut `pyclbr.Function`), 1658
`module d'extension`, 1756
`module_for_loader()` (dans le module `importlib.util`), 1635
`module_from_spec()` (dans le module `importlib.util`), 1635

- `module_repr()` (méthode `importlib.abc.Loader`), 1627
`ModuleFinder` (classe dans `modulefinder`), 1619
`modulefinder` (module), 1619
`ModuleInfo` (classe dans `pkgutil`), 1617
`ModuleNotFoundError`, 83
`modules` (attribut `modulefinder.ModuleFinder`), 1620
`modules` (dans le module `sys`), 1548
`ModuleSpec` (classe dans `importlib.machinery`), 1633
`ModuleType` (classe dans `types`), 240
`monotonic()` (dans le module `time`), 576
`month` (attribut `datetime.date`), 174
`month` (attribut `datetime.datetime`), 179
`month()` (dans le module `calendar`), 201
`month_abbr` (dans le module `calendar`), 201
`month_name` (dans le module `calendar`), 201
`monthcalendar()` (dans le module `calendar`), 201
`monthdatescalendar()` (méthode `calendar.Calendar`), 199
`monthdays2calendar()` (méthode `calendar.Calendar`), 199
`monthdayscalendar()` (méthode `calendar.Calendar`), 199
`monthrange()` (dans le module `calendar`), 201
`Morsel` (classe dans `http.cookies`), 1187
`most_common()` (méthode `collections.Counter`), 205
`mouseinterval()` (dans le module `curses`), 654
`mousemask()` (dans le module `curses`), 654
`move()` (dans le module `shutil`), 391
`move()` (méthode `curses.panel.Panel`), 672
`move()` (méthode `curses.window`), 661
`move()` (méthode `mmap.mmap`), 928
`move()` (méthode `tkinter.ttk.Treeview`), 1326
`move_to_end()` (méthode `collections.OrderedDict`), 215
`MozillaCookieJar` (classe dans `http.cookiejar`), 1193
MRO, 1760
`mro()` (méthode `class`), 79
`msg` (attribut `http.client.HTTPResponse`), 1136
`msg` (attribut `json.JSONDecodeError`), 991
`msg` (attribut `re.error`), 112
`msg` (attribut `traceback.TracebackException`), 1583
`msg()` (méthode `telnetlib.Telnet`), 1169
`msi`, 1683
`msilib` (module), 1683
`msvcrt` (module), 1689
`mt_interact()` (méthode `telnetlib.Telnet`), 1169
`mtime` (attribut `gzip.GzipFile`), 444
`mtime` (attribut `tarfile.TarInfo`), 466
`mtime()` (méthode `urllib.robotparser.RobotFileParser`), 1129
`muable`, 1760
`mul()` (dans le module `audioop`), 1225
`mul()` (dans le module `operator`), 344
`MultiCall` (classe dans `xmlrpc.client`), 1203
MULTILINE (dans le module `re`), 109
`MultipartConversionError`, 952
`multiply()` (méthode `decimal.Context`), 296
`multiprocessing` (module), 727
`multiprocessing.connection` (module), 755
`multiprocessing.dummy` (module), 759
`multiprocessing.Manager()` (dans le module `multiprocessing.sharedctypes`), 746
`multiprocessing.managers` (module), 746
`multiprocessing.pool` (module), 752
`multiprocessing.sharedctypes` (module), 744
`mutable`
 sequence types, 37
`MutableMapping` (classe dans `collections.abc`), 220
`MutableMapping` (classe dans `typing`), 1355
`MutableSequence` (classe dans `collections.abc`), 220
`MutableSequence` (classe dans `typing`), 1355
`MutableSet` (classe dans `collections.abc`), 220
`MutableSet` (classe dans `typing`), 1355
`mvderwin()` (méthode `curses.window`), 661
`mvwin()` (méthode `curses.window`), 661
`myrights()` (méthode `imaplib.IMAP4`), 1149
- ## N
- `-n N`
 timeit command line option, 1505
`n-uplet` nommé, 1760
`N_TOKENS` (dans le module `token`), 1651
`n_waiting` (attribut `threading.Barrier`), 726
`name` (attribut `codecs.CodecInfo`), 151
`name` (attribut `doctest.DocTest`), 1379
`name` (attribut `email.headerregistry.BaseHeader`), 953
`name` (attribut `hashlib.hash`), 505
`name` (attribut `hmac.HMAC`), 514
`name` (attribut `http.cookiejar.Cookie`), 1196
`name` (attribut `importlib.abc.FileLoader`), 1628
`name` (attribut `importlib.machinery.ExtensionFileLoader`), 1632
`name` (attribut `importlib.machinery.ModuleSpec`), 1633
`name` (attribut `importlib.machinery.SourceFileLoader`), 1632
`name` (attribut `importlib.machinery.SourcelessFileLoader`), 1632
`name` (attribut `inspect.Parameter`), 1597
`name` (attribut `io.FileIO`), 568
`name` (attribut `multiprocessing.Process`), 734
`name` (attribut `os.DirEntry`), 541
`name` (attribut `ossaudiodev.oss_audio_device`), 1240
`name` (attribut `pyclbr.Class`), 1658
`name` (attribut `pyclbr.Function`), 1658
`name` (attribut `tarfile.TarInfo`), 466
`name` (attribut `threading.Thread`), 718
`name` (attribut `xml.dom.Attr`), 1052
`name` (attribut `xml.dom.DocumentType`), 1050

- ul style="list-style-type: none; padding-left: 0;">
- `name` (dans le module `os`), 520
- `NAME` (dans le module `token`), 1651
- `name()` (dans le module `unicodedata`), 136
- `name2codepoint` (dans le module `html.entities`), 1028
- `NamedTemporaryFile()` (dans le module `tempfile`), 381
- `NamedTuple` (classe dans `typing`), 1358
- `namedtuple()` (dans le module `collections`), 212
- `NameError`, 83
- `namelist()` (méthode `zipfile.ZipFile`), 455
- `nameprep()` (dans le module `encodings.idna`), 166
- `namer` (attribut `logging.handlers.BaseRotatingHandler`), 640
- `namereplace_errors()` (dans le module `codecs`), 155
- `Namespace` (classe dans `argparse`), 604
- `Namespace` (classe dans `multiprocessing.managers`), 748
- `namespace()` (méthode `imaplib.IMAP4`), 1149
- `Namespace()` (méthode `multiprocessing.managers.SyncManager`), 748
- `NAMESPACE_DNS` (dans le module `uuid`), 1172
- `NAMESPACE_OID` (dans le module `uuid`), 1172
- `NAMESPACE_URL` (dans le module `uuid`), 1172
- `NAMESPACE_X500` (dans le module `uuid`), 1172
- `NamespaceErr`, 1054
- `namespaceURI` (attribut `xml.dom.Node`), 1048
- `NaN`, 11
- `nan` (dans le module `cmath`), 280
- `nan` (dans le module `math`), 277
- `nanj` (dans le module `cmath`), 280
- `NannyNag`, 1657
- `napms()` (dans le module `curses`), 654
- `nargs` (attribut `optparse.Option`), 1732
- `nbytes` (attribut `memoryview`), 69
- `ndiff()` (dans le module `difflib`), 124
- `ndim` (attribut `memoryview`), 70
- `ne` (2to3 fixer), 1471
- `ne()` (dans le module `operator`), 342
- `needs_input` (attribut `bz2.BZ2Decompressor`), 447
- `needs_input` (attribut `lzma.LZMADecompressor`), 451
- `neg()` (dans le module `operator`), 344
- `netmask` (attribut `ipaddress.IPv4Network`), 1216
- `netmask` (attribut `ipaddress.IPv6Network`), 1218
- `NetmaskValueError`, 1222
- `netrc` (classe dans `netrc`), 495
- `netrc` (module), 495
- `NetrcParseError`, 495
- `netscape` (attribut `http.cookiejar.CookiePolicy`), 1194
- `network` (attribut `ipaddress.IPv4Interface`), 1220
- `network` (attribut `ipaddress.IPv6Interface`), 1220
- `Network News Transfer Protocol`, 1152
- `network_address` (attribut `ipaddress.IPv4Network`), 1216
- `network_address` (attribut `ipaddress.IPv6Network`), 1218
- `new()` (dans le module `hashlib`), 504
- `new()` (dans le module `hmac`), 514
- `new_alignment()` (méthode `formatter.writer`), 1680
- `new_child()` (méthode `collections.ChainMap`), 202
- `new_class()` (dans le module `types`), 239
- `new_event_loop()` (dans le module `asyncio`), 870
- `new_event_loop()` (méthode `asyncio.AbstractEventLoopPolicy`), 872
- `new_font()` (méthode `formatter.writer`), 1680
- `new_margin()` (méthode `formatter.writer`), 1680
- `new_module()` (dans le module `imp`), 1746
- `new_panel()` (dans le module `curses.panel`), 671
- `new_spacing()` (méthode `formatter.writer`), 1680
- `new_styles()` (méthode `formatter.writer`), 1680
- `newgroups()` (méthode `nntplib.NNTP`), 1155
- `NEWLINE` (dans le module `token`), 1651
- `newlines` (attribut `io.TextIOBase`), 570
- `newnews()` (méthode `nntplib.NNTP`), 1155
- `newpad()` (dans le module `curses`), 654
- `NewType()` (dans le module `typing`), 1359
- `newwin()` (dans le module `curses`), 654
- `next` (2to3 fixer), 1471
- `next` (`pdb` command), 1493
- `next()` (fonction de base), 15
- `next()` (méthode `nntplib.NNTP`), 1157
- `next()` (méthode `tarfile.TarFile`), 464
- `next()` (méthode `tkinter.ttk.Treeview`), 1326
- `next_minus()` (méthode `decimal.Context`), 296
- `next_minus()` (méthode `decimal.Decimal`), 290
- `next_plus()` (méthode `decimal.Context`), 296
- `next_plus()` (méthode `decimal.Decimal`), 290
- `next_toward()` (méthode `decimal.Context`), 296
- `next_toward()` (méthode `decimal.Decimal`), 290
- `nextfile()` (dans le module `fileinput`), 373
- `nextkey()` (méthode `dbm.gnu.gdbm`), 417
- `nextSibling` (attribut `xml.dom.Node`), 1048
- `ngettext()` (dans le module `gettext`), 1244
- `ngettext()` (méthode `gettext.GNUTranslations`), 1247
- `ngettext()` (méthode `gettext.NullTranslations`), 1246
- `nice()` (dans le module `os`), 553
- `nis` (module), 1716
- `NL` (dans le module `tokenize`), 1653
- `nl()` (dans le module `curses`), 654
- `nl_langinfo()` (dans le module `locale`), 1254
- `nlargest()` (dans le module `heapq`), 223
- `nlst()` (méthode `ftplib.FTP`), 1142
- `NNTP`
 - protocol, 1152
- `NNTP` (classe dans `nntplib`), 1153
- `nntp_implementation` (attribut `nntplib.NNTP`), 1154
- `NNTP_SSL` (classe dans `nntplib`), 1153
- `nntp_version` (attribut `nntplib.NNTP`), 1154

- NNTPDataError, 1154
 NNTPError, 1153
 nntplib (module), 1152
 NNTPPermanentError, 1154
 NNTPProtocolError, 1154
 NNTPReplyError, 1154
 NNTPTemporaryError, 1154
 no_proxy, 1107
 no_type_check() (dans le module typing), 1360
 no_type_check_decorator() (dans le module typing), 1360
 nocbreak() (dans le module curses), 654
 NoDataAllowedErr, 1054
 node() (dans le module platform), 673
 nodelay() (méthode curses.window), 661
 nodeName (attribut xml.dom.Node), 1048
 NodeTransformer (classe dans ast), 1648
 nodeType (attribut xml.dom.Node), 1048
 nodeValue (attribut xml.dom.Node), 1048
 NodeVisitor (classe dans ast), 1647
 noecho() (dans le module curses), 654
 NOEXPR (dans le module locale), 1255
 nom qualifié, 1762
 nombre complexe, 1755
 nombre de références, 1763
 NoModificationAllowedErr, 1054
 nonblock() (méthode ossaudiodev.oss_audio_device), 1239
 NonCallableMagicMock (classe dans unittest.mock), 1438
 NonCallableMock (classe dans unittest.mock), 1423
 None (Built-in object), 27
 None (variable de base), 25
 nonl() (dans le module curses), 654
 nonzero (2to3 fixer), 1471
 noop() (méthode imaplib.IMAP4), 1149
 noop() (méthode poplib.POP3), 1145
 NoOptionError, 494
 NOP (opcode), 1666
 noqiflush() (dans le module curses), 654
 noraw() (dans le module curses), 655
 --no-report
 trace command line option, 1509
 NoReturn (dans le module typing), 1360
 normalize() (dans le module locale), 1256
 normalize() (dans le module unicodedata), 136
 normalize() (méthode decimal.Context), 296
 normalize() (méthode decimal.Decimal), 290
 normalize() (méthode xml.dom.Node), 1049
 NORMALIZE_WHITESPACE (dans le module doctest), 1371
 normalvariate() (dans le module random), 312
 normcase() (dans le module os.path), 370
 normpath() (dans le module os.path), 370
 NoSectionError, 494
 NoSuchMailboxError, 1011
 not
 opérateur, 28
 not in
 opérateur, 28, 35
 not_() (dans le module operator), 343
 NotADirectoryError, 88
 notationDecl() (méthode xml.sax.handler.DTDHandler), 1068
 NotationDeclHandler() (méthode xml.parsers.expat.xmlparser), 1077
 notations (attribut xml.dom.DocumentType), 1050
 NotConnected, 1133
 Notebook (classe dans tkinter.tix), 1333
 Notebook (classe dans tkinter.ttk), 1320
 NotEmptyError, 1011
 NOTEQUAL (dans le module token), 1651
 NotFoundErr, 1054
 notify() (méthode asyncio.Condition), 906
 notify() (méthode threading.Condition), 722
 notify_all() (méthode asyncio.Condition), 906
 notify_all() (méthode threading.Condition), 722
 notimeout() (méthode curses.window), 661
 NotImplemented (variable de base), 25
 NotImplementedError, 84
 NotStandaloneHandler() (méthode xml.parsers.expat.xmlparser), 1078
 NotSupportedErr, 1054
 NotSupportedError, 431
 noutrefresh() (méthode curses.window), 661
 nouvelle classe, 1761
 now() (méthode de la classe datetime.datetime), 177
 NSIG (dans le module signal), 922
 nsmallest() (dans le module heapq), 223
 NT_OFFSET (dans le module token), 1651
 NTEventLogHandler (classe dans logging.handlers), 646
 ntohl() (dans le module socket), 804
 ntohs() (dans le module socket), 805
 ntransfercmd() (méthode ftplib.FTP), 1142
 NullFormatter (classe dans formatter), 1679
 NullHandler (classe dans logging), 639
 NullImporter (classe dans imp), 1748
 NullTranslations (classe dans gettext), 1246
 NullWriter (classe dans formatter), 1681
 num_addresses (attribut ipaddress.IPv4Network), 1216
 num_addresses (attribut ipaddress.IPv6Network), 1218
 Number (classe dans numbers), 269
 NUMBER (dans le module token), 1651
 --number=N
 timeit command line option, 1505
 number_class() (méthode decimal.Context), 296
 number_class() (méthode decimal.Decimal), 290

numbers (*module*), 269
 numerator (*attribut fractions.Fraction*), 307
 numerator (*attribut numbers.Rational*), 270
 numeric
 conversions, 30
 literals, 29
 object, 28
 objet, 29
 types, operations on, 29
 numeric() (*dans le module unicodedata*), 136
 Numerical Python, 20
 numinput() (*dans le module turtle*), 1285
 numliterals (*2to3 fixer*), 1472

O

-o
 pickletools command line option, 1675
 -o <output>
 zipapp command line option, 1532
 O_APPEND (*dans le module os*), 529
 O_ASYNC (*dans le module os*), 529
 O_BINARY (*dans le module os*), 529
 O_CLOEXEC (*dans le module os*), 529
 O_CREAT (*dans le module os*), 529
 O_DIRECT (*dans le module os*), 529
 O_DIRECTORY (*dans le module os*), 529
 O_DSYNC (*dans le module os*), 529
 O_EXCL (*dans le module os*), 529
 O_EXLOCK (*dans le module os*), 529
 O_NDELAY (*dans le module os*), 529
 O_NOATIME (*dans le module os*), 529
 O_NOCTTY (*dans le module os*), 529
 O_NOFOLLOW (*dans le module os*), 529
 O_NOINHERIT (*dans le module os*), 529
 O_NONBLOCK (*dans le module os*), 529
 O_PATH (*dans le module os*), 529
 O_RANDOM (*dans le module os*), 529
 O_RDONLY (*dans le module os*), 529
 O_RDWR (*dans le module os*), 529
 O_RSYNC (*dans le module os*), 529
 O_SEQUENTIAL (*dans le module os*), 529
 O_SHLOCK (*dans le module os*), 529
 O_SHORT_LIVED (*dans le module os*), 529
 O_SYNC (*dans le module os*), 529
 O_TEMPORARY (*dans le module os*), 529
 O_TEXT (*dans le module os*), 529
 O_TMPFILE (*dans le module os*), 529
 O_TRUNC (*dans le module os*), 529
 O_WRONLY (*dans le module os*), 529
 obj (*attribut memoryview*), 69
 object
 code, 78, 414
 numeric, 28

object (*attribut UnicodeError*), 86
 object (*classe de base*), 15
 objects
 comparing, 28
 flattening, 397
 marshalling, 397
 persistent, 397
 pickling, 397
 serializing, 397
 objet, 1761
 Boolean, 29
 bytearray, 37, 51, 52
 bytes, 51
 complex number, 29
 dictionary, 73
 floating point, 29
 integer, 29
 io.StringIO, 41
 list, 37, 38
 mapping, 73
 memoryview, 51
 method, 77
 numeric, 29
 range, 40
 sequence, 35
 set, 70
 socket, 797
 string, 41
 traceback, 1541, 1581
 tuple, 37, 39
 type, 22
 objet fichier, 1756
 objet fichier-compatible, 1757
 objet octet-compatible, 1754
 objet simili-chemin, 1761
 obufcount() (*méthode ossaudiodev.oss_audio_device*), 1240
 obuffree() (*méthode ossaudiodev.oss_audio_device*), 1240
 oct() (*fonction de base*), 15
 octal
 literals, 29
 octdigits (*dans le module string*), 92
 offset (*attribut traceback.TracebackException*), 1583
 offset (*attribut xml.parsers.expat.ExpatError*), 1079
 OK (*dans le module curses*), 663
 OleDLL (*classe dans ctypes*), 701
 onclick() (*dans le module turtle*), 1279, 1284
 ondrag() (*dans le module turtle*), 1279
 onecmd() (*méthode cmd.Cmd*), 1294
 onkey() (*dans le module turtle*), 1283
 onkeypress() (*dans le module turtle*), 1284
 onkeyrelease() (*dans le module turtle*), 1283
 onrelease() (*dans le module turtle*), 1279

- `onscreenclick()` (dans le module `turtle`), 1284
- `ontimer()` (dans le module `turtle`), 1284
- `OP` (dans le module `token`), 1651
- `OP_ALL` (dans le module `ssl`), 825
- `OP_CIPHER_SERVER_PREFERENCE` (dans le module `ssl`), 826
- `OP_ENABLE_MIDDLEBOX_COMPAT` (dans le module `ssl`), 826
- `OP_NO_COMPRESSION` (dans le module `ssl`), 826
- `OP_NO_SSLv2` (dans le module `ssl`), 825
- `OP_NO_SSLv3` (dans le module `ssl`), 826
- `OP_NO_TICKET` (dans le module `ssl`), 826
- `OP_NO_TLSv1` (dans le module `ssl`), 826
- `OP_NO_TLSv1_1` (dans le module `ssl`), 826
- `OP_NO_TLSv1_2` (dans le module `ssl`), 826
- `OP_NO_TLSv1_3` (dans le module `ssl`), 826
- `OP_SINGLE_DH_USE` (dans le module `ssl`), 826
- `OP_SINGLE_ECDH_USE` (dans le module `ssl`), 826
- `open()` (dans le module `aifc`), 1226
- `open()` (dans le module `bz2`), 446
- `open()` (dans le module `codecs`), 152
- `open()` (dans le module `dbm`), 415
- `open()` (dans le module `dbm.dumb`), 418
- `open()` (dans le module `dbm.gnu`), 416
- `open()` (dans le module `dbm.ndbm`), 417
- `open()` (dans le module `gzip`), 443
- `open()` (dans le module `io`), 564
- `open()` (dans le module `lzma`), 448
- `open()` (dans le module `os`), 528
- `open()` (dans le module `ossaudiodev`), 1237
- `open()` (dans le module `shelve`), 411
- `open()` (dans le module `sunau`), 1229
- `open()` (dans le module `tarfile`), 461
- `open()` (dans le module `tokenize`), 1654
- `open()` (dans le module `wave`), 1231
- `open()` (dans le module `webbrowser`), 1086
- `open()` (fonction de base), 15
- `open()` (méthode de la classe `tarfile.TarFile`), 464
- `open()` (méthode `imaplib.IMAP4`), 1149
- `open()` (méthode `pathlib.Path`), 364
- `open()` (méthode `pipes.Template`), 1712
- `open()` (méthode `telnetlib.Telnet`), 1169
- `open()` (méthode `urllib.request.OpenerDirector`), 1110
- `open()` (méthode `urllib.request.URLopener`), 1120
- `open()` (méthode `webbrowser.controller`), 1088
- `open()` (méthode `zipfile.ZipFile`), 455
- `open_connection()` (dans le module `asyncio`), 892
- `open_new()` (dans le module `webbrowser`), 1086
- `open_new()` (méthode `webbrowser.controller`), 1088
- `open_new_tab()` (dans le module `webbrowser`), 1086
- `open_new_tab()` (méthode `webbrowser.controller`), 1088
- `open_osfhandle()` (dans le module `msvcrt`), 1689
- `open_unix_connection()` (dans le module `asyncio`), 893
- `open_unknown()` (méthode `urllib.request.URLopener`), 1120
- `OpenDatabase()` (dans le module `msilib`), 1683
- `OpenerDirector` (classe dans `urllib.request`), 1107
- `openfp()` (dans le module `sunau`), 1229
- `openfp()` (dans le module `wave`), 1232
- `OpenKey()` (dans le module `winreg`), 1693
- `OpenKeyEx()` (dans le module `winreg`), 1693
- `openlog()` (dans le module `syslog`), 1717
- `openmixer()` (dans le module `ossaudiodev`), 1238
- `openpty()` (dans le module `os`), 529
- `openpty()` (dans le module `pty`), 1708
- `OpenSSL`
 - (use in module `hashlib`), 504
 - (use in module `ssl`), 817
- `OPENSSL_VERSION` (dans le module `ssl`), 827
- `OPENSSL_VERSION_INFO` (dans le module `ssl`), 827
- `OPENSSL_VERSION_NUMBER` (dans le module `ssl`), 827
- `OpenView()` (méthode `msilib.Database`), 1684
- opérateur
 - `%` (percent), 29
 - `&` (ampersand), 30
 - `*` (asterisk), 29
 - `**`, 29
 - `/` (slash), 29
 - `//`, 29
 - `<` (less), 28
 - `<<`, 30
 - `<=`, 28
 - `!=`, 28
 - `==`, 28
 - `>` (greater), 28
 - `>=`, 28
 - `>>`, 30
 - `^` (caret), 30
 - `|` (vertical bar), 30
 - `~` (tilde), 30
 - `and`, 27, 28
 - `in`, 28, 35
 - `is`, 28
 - `is not`, 28
 - `not`, 28
 - `not in`, 28, 35
 - `or`, 27, 28
- operation
 - concatenation, 35
 - repetition, 35
 - slice, 35
 - subscript, 35
- `OperationalError`, 431
- operations
 - bitwise, 30

Boolean, 27, 28
 masking, 30
 shifting, 30
 operations on
 dictionary type, 73
 integer types, 30
 list type, 37
 mapping types, 73
 numeric types, 29
 sequence types, 35, 37
 operator
 - (*minus*), 29
 + (*plus*), 29
 comparison, 28
 operator (2to3 *fixer*), 1472
 operator (*module*), 342
 opmap (*dans le module dis*), 1674
 opname (*dans le module dis*), 1674
 optimize() (*dans le module pickletools*), 1676
 OPTIMIZED_BYTECODE_SUFFIXES (*dans le module importlib.machinery*), 1630
 Optional (*dans le module typing*), 1361
 OptionGroup (*classe dans optparse*), 1726
 OptionMenu (*classe dans tkinter.tix*), 1332
 OptionParser (*classe dans optparse*), 1729
 options (*attribut doctest.Example*), 1379
 options (*attribut ssl.SSLContext*), 836
 Options (*classe dans ssl*), 826
 options() (*méthode configparser.ConfigParser*), 491
 optionxform() (*méthode configparser.ConfigParser*), 487, 492
 optparse (*module*), 1719
 or
 opérateur, 27, 28
 or_() (*dans le module operator*), 344
 ord() (*fonction de base*), 18
 ordered_attributes (*attribut xml.parsers.expat.xmlparser*), 1076
 OrderedDict (*classe dans collections*), 215
 ordre de résolution des méthodes, 1760
 origin (*attribut importlib.machinery.ModuleSpec*), 1633
 origin_req_host (*attribut urllib.request.Request*), 1109
 origin_server (*attribut wsgi-ref.handlers.BaseHandler*), 1103
 os
 module, 1701
 os (*module*), 519
 os_environ (*attribut wsgiref.handlers.BaseHandler*), 1102
 OSError, 84
 os.path (*module*), 367
 ossaudiodev (*module*), 1237
 OSSAudioError, 1237

outfile
 json.tool command line option, 994
 output (*attribut subprocess.CalledProcessError*), 776
 output (*attribut subprocess.TimeoutExpired*), 776
 output (*attribut unittest.TestCase*), 1400
 output() (*méthode http.cookies.BaseCookie*), 1187
 output() (*méthode http.cookies.Morsel*), 1188
 --output=<file>
 pickletools command line option, 1675
 --output=<output>
 zipapp command line option, 1532
 output_charset (*attribut email.charset.Charset*), 979
 output_charset() (*méthode get-text.NullTranslations*), 1246
 output_codec (*attribut email.charset.Charset*), 980
 output_difference() (*méthode doc-test.OutputChecker*), 1382
 OutputChecker (*classe dans doctest*), 1382
 OutputString() (*méthode http.cookies.Morsel*), 1188
 over() (*méthode nntplib.NNTP*), 1156
 Overflow (*classe dans decimal*), 298
 OverflowError, 84
 overlaps() (*méthode ipaddress.IPv4Network*), 1216
 overlaps() (*méthode ipaddress.IPv6Network*), 1218
 overlay() (*méthode curses.window*), 661
 overload() (*dans le module typing*), 1359
 overwrite() (*méthode curses.window*), 661
 owner() (*méthode pathlib.Path*), 365

P

-p
 pickletools command line option, 1675
 timeit command line option, 1505
 unittest-discover command line option, 1389
 p (*pdb command*), 1494
 -p <interpreter>
 zipapp command line option, 1532
 P_ALL (*dans le module os*), 556
 P_DETACH (*dans le module os*), 555
 P_NOWAIT (*dans le module os*), 555
 P_NOWAITO (*dans le module os*), 555
 P_OVERLAY (*dans le module os*), 555
 P_PGID (*dans le module os*), 556
 P_PID (*dans le module os*), 556
 P_WAIT (*dans le module os*), 555
 pack() (*dans le module struct*), 146
 pack() (*méthode mailbox.MH*), 1001
 pack() (*méthode struct.Struct*), 150
 pack_array() (*méthode xdrlib.Packer*), 497
 pack_bytes() (*méthode xdrlib.Packer*), 497
 pack_double() (*méthode xdrlib.Packer*), 496

- `pack_farray()` (méthode `xdrlib.Packer`), 497
- `pack_float()` (méthode `xdrlib.Packer`), 496
- `pack_fopaque()` (méthode `xdrlib.Packer`), 496
- `pack_fstring()` (méthode `xdrlib.Packer`), 496
- `pack_into()` (dans le module `struct`), 146
- `pack_into()` (méthode `struct.Struct`), 150
- `pack_list()` (méthode `xdrlib.Packer`), 497
- `pack_opaque()` (méthode `xdrlib.Packer`), 496
- `pack_string()` (méthode `xdrlib.Packer`), 496
- `package`, 1606
- `packed` (attribut `ipaddress.IPv4Address`), 1212
- `packed` (attribut `ipaddress.IPv6Address`), 1213
- `Packer` (classe dans `xdrlib`), 496
- `packing`
 - binary data, 145
- `packing` (widgets), 1309
- `PAGER`, 1363
- `pair_content()` (dans le module `curses`), 655
- `pair_number()` (dans le module `curses`), 655
- `PanedWindow` (classe dans `tkinter.tix`), 1333
- `paquet`, 1761
- `paquet classique`, 1763
- `paquet provisoire`, 1762
- `paquet-espace de nommage`, 1760
- `Parameter` (classe dans `inspect`), 1597
- `ParameterizedMIMEHeader` (classe dans `email.headerregistry`), 956
- `parameters` (attribut `inspect.Signature`), 1596
- `paramètre`, 1761
- `params` (attribut `email.headerregistry.ParameterizedMIMEHeader`), 956
- `pardir` (dans le module `os`), 560
- `paren` (2to3 fixer), 1472
- `parent` (attribut `importlib.machinery.ModuleSpec`), 1633
- `parent` (attribut `urllib.request.BaseHandler`), 1111
- `parent()` (méthode `tkinter.ttk.Treeview`), 1326
- `parentNode` (attribut `xml.dom.Node`), 1048
- `parents` (attribut `collections.ChainMap`), 203
- `paretovariate()` (dans le module `random`), 312
- `parse()` (dans le module `ast`), 1647
- `parse()` (dans le module `cgi`), 1091
- `parse()` (dans le module `xml.dom.minidom`), 1056
- `parse()` (dans le module `xml.dom.pulldom`), 1061
- `parse()` (dans le module `xml.etree.ElementTree`), 1038
- `parse()` (dans le module `xml.sax`), 1062
- `parse()` (méthode `doctest.DocTestParser`), 1380
- `parse()` (méthode `email.parser.BytesParser`), 941
- `parse()` (méthode `email.parser.Parser`), 942
- `parse()` (méthode `string.Formatter`), 92
- `parse()` (méthode `urllib.robotparser.RobotFileParser`), 1129
- `parse()` (méthode `xml.etree.ElementTree.ElementTree`), 1041
- `Parse()` (méthode `xml.parsers.expat.xmlparser`), 1075
- `parse()` (méthode `xml.sax.xmlreader.XMLReader`), 1071
- `parse_and_bind()` (dans le module `readline`), 140
- `parse_args()` (méthode `argparse.ArgumentParser`), 601
- `PARSE_COLNAMES` (dans le module `sqlite3`), 421
- `parse_config_h()` (dans le module `sysconfig`), 1556
- `PARSE_DECLTYPES` (dans le module `sqlite3`), 420
- `parse_header()` (dans le module `cgi`), 1092
- `parse_known_args()` (méthode `argparse.ArgumentParser`), 610
- `parse_multipart()` (dans le module `cgi`), 1091
- `parse_qs()` (dans le module `cgi`), 1091
- `parse_qs()` (dans le module `urllib.parse`), 1123
- `parse_qsl()` (dans le module `cgi`), 1091
- `parse_qsl()` (dans le module `urllib.parse`), 1123
- `parseaddr()` (dans le module `email.utils`), 982
- `parsebytes()` (méthode `email.parser.BytesParser`), 941
- `parsedate()` (dans le module `email.utils`), 983
- `parsedate_to_datetime()` (dans le module `email.utils`), 983
- `parsedate_tz()` (dans le module `email.utils`), 983
- `ParseError` (classe dans `xml.etree.ElementTree`), 1045
- `ParseFile()` (méthode `xml.parsers.expat.xmlparser`), 1075
- `ParseFlags()` (dans le module `imaplib`), 1147
- `Parser` (classe dans `email.parser`), 941
- `parser` (module), 1639
- `ParserCreate()` (dans le module `xml.parsers.expat`), 1074
- `ParserError`, 1642
- `ParseResult` (classe dans `urllib.parse`), 1126
- `ParseResultBytes` (classe dans `urllib.parse`), 1126
- `parsestr()` (méthode `email.parser.Parser`), 942
- `parseString()` (dans le module `xml.dom.minidom`), 1056
- `parseString()` (dans le module `xml.dom.pulldom`), 1061
- `parseString()` (dans le module `xml.sax`), 1062
- `parsing`
 - Python source code, 1639
 - URL, 1122
- `ParsingError`, 494
- `partial` (attribut `asyncio.IncompleteReadError`), 895
- `partial()` (dans le module `functools`), 338
- `partial()` (méthode `imaplib.IMAP4`), 1149
- `partialmethod` (classe dans `functools`), 338
- `parties` (attribut `threading.Barrier`), 726
- `partition()` (méthode `bytearray`), 54
- `partition()` (méthode `bytes`), 54
- `partition()` (méthode `str`), 45
- `pass_()` (méthode `poplib.POP3`), 1144
- `Paste`, 1339
- `patch()` (dans le module `unittest.mock`), 1428

- `patch.dict()` (dans le module `unittest.mock`), 1431
- `patch.multiple()` (dans le module `unittest.mock`), 1432
- `patch.object()` (dans le module `unittest.mock`), 1431
- `patch.stopall()` (dans le module `unittest.mock`), 1434
- `PATH`, 551, 554, 561, 1085, 1093, 1094
- `path`
 - configuration file, 1606
 - module search, 387, 1548, 1605
 - operations, 351, 367
- `path` (attribut `http.cookiejar.Cookie`), 1196
- `path` (attribut `http.server.BaseHTTPRequestHandler`), 1182
- `path` (attribut `importlib.abc.FileLoader`), 1628
- `path` (attribut `importlib.machinery.ExtensionFileLoader`), 1633
- `path` (attribut `importlib.machinery.FileFinder`), 1631
- `path` (attribut `importlib.machinery.SourceFileLoader`), 1632
- `path` (attribut `importlib.machinery.SourcelessFileLoader`), 1632
- `path` (attribut `os.DirEntry`), 541
- `Path` (classe dans `pathlib`), 361
- `path` (dans le module `sys`), 1548
- `Path browser`, 1336
- `path_hook()` (méthode de la classe `importlib.machinery.FileFinder`), 1632
- `path_hooks` (dans le module `sys`), 1548
- `path_importer_cache` (dans le module `sys`), 1548
- `path_mtime()` (méthode `importlib.abc.SourceLoader`), 1629
- `path_return_ok()` (méthode `http.cookiejar.CookiePolicy`), 1194
- `path_stats()` (méthode `importlib.abc.SourceLoader`), 1629
- `path_stats()` (méthode `importlib.machinery.SourceFileLoader`), 1632
- `pathconf()` (dans le module `os`), 538
- `pathconf_names` (dans le module `os`), 539
- `PathEntryFinder` (classe dans `importlib.abc`), 1626
- `PathFinder` (classe dans `importlib.machinery`), 1631
- `pathlib` (module), 351
- `PathLike` (classe dans `os`), 521
- `pathname2url()` (dans le module `urllib.request`), 1106
- `pathsep` (dans le module `os`), 561
- `pattern` (attribut `re.error`), 112
- `pattern` (attribut `re.regex`), 114
- `Pattern` (classe dans `typing`), 1358
- `--pattern pattern`
 - `unittest-discover` command line option, 1389
- `pause()` (dans le module `signal`), 923
- `pause_reading()` (méthode `asyncio.ReadTransport`), 884
- `pause_writing()` (méthode `asyncio.BaseProtocol`), 887
- `PAX_FORMAT` (dans le module `tarfile`), 463
- `pax_headers` (attribut `tarfile.TarFile`), 465
- `pax_headers` (attribut `tarfile.TarInfo`), 466
- `pbkdf2_hmac()` (dans le module `hashlib`), 506
- `pd()` (dans le module `turtle`), 1272
- `Pdb` (class in `pdb`), 1489
- `Pdb` (classe dans `pdb`), 1491
- `pdb` (module), 1489
- `.pdbrc`
 - file, 1492
- `peek()` (méthode `bz2.BZ2File`), 446
- `peek()` (méthode `gzip.GzipFile`), 444
- `peek()` (méthode `io.BufferedReader`), 569
- `peek()` (méthode `lzma.LZMAFile`), 449
- `peek()` (méthode `weakref.finalize`), 234
- `peer` (attribut `smtpd.SMTPChannel`), 1167
- `PEM_cert_to_DER_cert()` (dans le module `ssl`), 822
- `pen()` (dans le module `turtle`), 1272
- `pencolor()` (dans le module `turtle`), 1273
- `pending` (attribut `ssl.MemoryBIO`), 844
- `pending()` (méthode `ssl.SSLSocket`), 831
- `PendingDeprecationWarning`, 88
- `pendown()` (dans le module `turtle`), 1272
- `pensize()` (dans le module `turtle`), 1272
- `penup()` (dans le module `turtle`), 1272
- `PEP`, 1762
- `PERCENT` (dans le module `token`), 1651
- `PERCENTEQUAL` (dans le module `token`), 1651
- `perf_counter()` (dans le module `time`), 576
- `Performance`, 1503
- `PermissionError`, 88
- `permutations()` (dans le module `itertools`), 328
- `Persist()` (méthode `msilib.SummaryInformation`), 1685
- `persistence`, 397
- `persistent`
 - objects, 397
- `persistent_id` (`pickle protocol`), 404
- `persistent_id()` (méthode `pickle.Pickler`), 400
- `persistent_load` (`pickle protocol`), 404
- `persistent_load()` (méthode `pickle.Unpickler`), 401
- `PF_CAN` (dans le module `socket`), 801
- `PF_PACKET` (dans le module `socket`), 801
- `PF_RDS` (dans le module `socket`), 801
- `pformat()` (dans le module `pprint`), 244
- `pformat()` (méthode `pprint.PrettyPrinter`), 245
- `phase()` (dans le module `cmath`), 278
- `pi` (dans le module `cmath`), 280
- `pi` (dans le module `math`), 277
- `pickle`
 - module, 243, 410, 411, 413

- `pickle` (*module*), 397
- `pickle()` (*dans le module copyreg*), 410
- `PickleError`, 400
- `Pickler` (*classe dans pickle*), 400
- `pickletools` (*module*), 1675
- `pickletools` command line option
 - `-a`, 1675
 - `--annotate`, 1675
 - `--indentlevel=<num>`, 1675
 - `-l`, 1675
 - `-m`, 1675
 - `--memo`, 1675
 - `-o`, 1675
 - `--output=<file>`, 1675
 - `-p`, 1675
 - `--preamble=<preamble>`, 1675
- `pickling`
 - objects, 397
- `PicklingError`, 400
- `pid` (*attribut asyncio.asyncio.subprocess.Process*), 901
- `pid` (*attribut multiprocessing.Process*), 734
- `pid` (*attribut subprocess.Popen*), 782
- `PIPE` (*dans le module subprocess*), 775
- `Pipe()` (*dans le module multiprocessing*), 736
- `pipe()` (*dans le module os*), 530
- `pipe2()` (*dans le module os*), 530
- `PIPE_BUF` (*dans le module select*), 848
- `pipe_connection_lost()` (*méthode asyn-*
cio.SubprocessProtocol), 886
- `pipe_data_received()` (*méthode asyn-*
cio.SubprocessProtocol), 886
- `pipes` (*module*), 1711
- `PKG_DIRECTORY` (*dans le module imp*), 1748
- `pkgutil` (*module*), 1617
- `placeholder` (*attribut textwrap.TextWrapper*), 135
- `platform` (*dans le module sys*), 1548
- `platform` (*module*), 672
- `platform()` (*dans le module platform*), 673
- `PlaySound()` (*dans le module winsound*), 1698
- `plist`
 - file, 499
- `plistlib` (*module*), 499
- `plock()` (*dans le module os*), 553
- `PLUS` (*dans le module token*), 1651
- `plus()` (*méthode decimal.Context*), 296
- `PLUSEQUAL` (*dans le module token*), 1651
- `pm()` (*dans le module pdb*), 1491
- `point d'entrée pour la recherche dans`
`path`, 1761
- `POINTER()` (*dans le module ctypes*), 707
- `pointer()` (*dans le module ctypes*), 707
- `polar()` (*dans le module cmath*), 278
- `Policy` (*classe dans email.policy*), 947
- `poll()` (*dans le module select*), 847
- `poll()` (*méthode multiprocessing.connection.Connection*),
740
- `poll()` (*méthode select.devpoll*), 849
- `poll()` (*méthode select.epoll*), 850
- `poll()` (*méthode select.poll*), 851
- `poll()` (*méthode subprocess.Popen*), 780
- `PollSelector` (*classe dans selectors*), 856
- `Pool` (*classe dans multiprocessing.pool*), 752
- `pop()` (*méthode array.array*), 230
- `pop()` (*méthode collections.deque*), 208
- `pop()` (*méthode dict*), 74
- `pop()` (*méthode frozenset*), 72
- `pop()` (*méthode mailbox.Mailbox*), 998
- `pop()` (*sequence method*), 37
- `POP3`
 - protocol, 1143
- `POP3` (*classe dans poplib*), 1143
- `POP3_SSL` (*classe dans poplib*), 1143
- `pop_alignment()` (*méthode formatter.formatter*),
1678
- `pop_all()` (*méthode contextlib.ExitStack*), 1568
- `POP_BLOCK` (*opcode*), 1669
- `POP_EXCEPT` (*opcode*), 1669
- `pop_font()` (*méthode formatter.formatter*), 1679
- `POP_JUMP_IF_FALSE` (*opcode*), 1671
- `POP_JUMP_IF_TRUE` (*opcode*), 1671
- `pop_margin()` (*méthode formatter.formatter*), 1679
- `pop_source()` (*méthode shlex.shlex*), 1300
- `pop_style()` (*méthode formatter.formatter*), 1679
- `POP_TOP` (*opcode*), 1666
- `Popen` (*classe dans subprocess*), 777
- `popen()` (*dans le module os*), 553
- `popen()` (*dans le module platform*), 674
- `popen()` (*in module os*), 848
- `popitem()` (*méthode collections.OrderedDict*), 215
- `popitem()` (*méthode dict*), 74
- `popitem()` (*méthode mailbox.Mailbox*), 998
- `popleft()` (*méthode collections.deque*), 208
- `poplib` (*module*), 1143
- `PopupMenu` (*classe dans tkinter.tix*), 1332
- `port` (*attribut http.cookiejar.Cookie*), 1196
- `port_specified` (*attribut http.cookiejar.Cookie*), 1197
- `portée imbriquée`, 1760
- `portion`, 1762
- `pos` (*attribut json.JSONDecodeError*), 991
- `pos` (*attribut re.error*), 112
- `pos` (*attribut re.match*), 116
- `pos()` (*dans le module operator*), 344
- `pos()` (*dans le module turtle*), 1270
- `position` (*attribut xml.etree.ElementTree.ParseError*),
1045
- `position()` (*dans le module turtle*), 1270
- `POSIX`
 - I/O control, 1706

- threads, 794
- posix (module), 1701
- POSIX_FADV_DONTNEED (dans le module os), 530
- POSIX_FADV_NOREUSE (dans le module os), 530
- POSIX_FADV_NORMAL (dans le module os), 530
- POSIX_FADV_RANDOM (dans le module os), 530
- POSIX_FADV_SEQUENTIAL (dans le module os), 530
- POSIX_FADV_WILLNEED (dans le module os), 530
- posix_fadvise() (dans le module os), 530
- posix_fallocate() (dans le module os), 530
- POSIXLY_CORRECT, 612
- PosixPath (classe dans pathlib), 361
- post() (méthode nntplib.NNTP), 1157
- post() (méthode ossaudiodev.oss_audio_device), 1239
- post_handshake_auth (attribut ssl.SSLContext), 837
- post_mortem() (dans le module pdb), 1490
- post_setup() (méthode venv.EnvBuilder), 1527
- postcmd() (méthode cmd.Cmd), 1294
- postloop() (méthode cmd.Cmd), 1294
- pow() (dans le module math), 275
- pow() (dans le module operator), 344
- pow() (fonction de base), 18
- power() (méthode decimal.Context), 296
- pp (pdb command), 1494
- pprint (module), 243
- pprint() (dans le module pprint), 244
- pprint() (méthode pprint.PrettyPrinter), 245
- prcal() (dans le module calendar), 201
- pread() (dans le module os), 530
- preamble (attribut email.message.EmailMessage), 939
- preamble (attribut email.message.Message), 974
- preamble=<preamble>
 - pickletools command line option, 1675
- precmd() (méthode cmd.Cmd), 1294
- prefix (attribut xml.dom.Attr), 1052
- prefix (attribut xml.dom.Node), 1048
- prefix (attribut zipimport.zipimporter), 1616
- prefix (dans le module sys), 1549
- PREFIXES (dans le module site), 1607
- prefixlen (attribut ipaddress.IPv4Network), 1216
- prefixlen (attribut ipaddress.IPv6Network), 1218
- preloop() (méthode cmd.Cmd), 1294
- prepare() (méthode logging.handlers.QueueHandler), 648
- prepare() (méthode logging.handlers.QueueListener), 649
- prepare_class() (dans le module types), 239
- prepare_input_source() (dans le module xml.sax.saxutils), 1069
- prepend() (méthode pipes.Template), 1712
- PrettyPrinter (classe dans pprint), 243
- prev() (méthode tkinter.ttk.Treeview), 1326
- previousSibling (attribut xml.dom.Node), 1048
- print (2to3 fixer), 1472
- print() (fonction de base), 18
- print_callees() (méthode pstats.Stats), 1501
- print_callers() (méthode pstats.Stats), 1500
- print_directory() (dans le module cgi), 1092
- print_environ() (dans le module cgi), 1092
- print_environ_usage() (dans le module cgi), 1092
- print_exc() (dans le module traceback), 1581
- print_exc() (méthode timeit.Timer), 1505
- print_exception() (dans le module traceback), 1581
- PRINT_EXPR (opcode), 1668
- print_form() (dans le module cgi), 1092
- print_help() (méthode argparse.ArgumentParser), 610
- print_last() (dans le module traceback), 1581
- print_stack() (dans le module traceback), 1581
- print_stack() (méthode asyncio.Task), 879
- print_stats() (méthode profile.Profile), 1498
- print_stats() (méthode pstats.Stats), 1500
- print_tb() (dans le module traceback), 1581
- print_usage() (méthode argparse.ArgumentParser), 610
- print_usage() (méthode optparse.OptionParser), 1738
- print_version() (méthode optparse.OptionParser), 1728
- printable (dans le module string), 92
- printdir() (méthode zipfile.ZipFile), 456
- printf-style formatting, 49, 62
- PRIO_GRP (dans le module os), 523
- PRIO_PROCESS (dans le module os), 523
- PRIO_USER (dans le module os), 523
- PriorityQueue (classe dans asyncio), 908
- PriorityQueue (classe dans queue), 791
- prlimit() (dans le module resource), 1713
- prmonth() (dans le module calendar), 201
- prmonth() (méthode calendar.TextCalendar), 200
- ProactorEventLoop (classe dans asyncio), 870
- process
 - group, 522
 - id, 523
 - id of parent, 523
 - killing, 553
 - scheduling priority, 523, 524
 - signalling, 553
- process
 - timeit command line option, 1505
- Process (classe dans multiprocessing), 733
- process() (méthode logging.LoggerAdapter), 624
- process_exited() (méthode asyncio.SubprocessProtocol), 886
- process_message() (méthode smtpd.SMTPServer), 1165

- `process_request()` (méthode `socketserver.BaseServer`), 1176
`process_time()` (dans le module `time`), 576
`process_tokens()` (dans le module `tabnanny`), 1657
`ProcessError`, 735
`processes`, light-weight, 794
`ProcessingInstruction()` (dans le module `xml.etree.ElementTree`), 1038
`processingInstruction()` (méthode `xml.sax.handler.ContentHandler`), 1067
`ProcessingInstructionHandler()` (méthode `xml.parsers.expat.xmlparser`), 1077
`ProcessLookupError`, 88
`processor time`, 574
`processor()` (dans le module `platform`), 673
`ProcessPoolExecutor` (classe dans `concurrent.futures`), 771
`product()` (dans le module `itertools`), 329
`Profile` (classe dans `profile`), 1498
`profile` (module), 1498
`profile function`, 716, 1545, 1550
`profiler`, 1545, 1550
`profiling`, deterministic, 1495
`ProgrammingError`, 431
`Progressbar` (classe dans `tkinter.ttk`), 1322
`prompt` (attribut `cmd.Cmd`), 1294
`prompt_user_passwd()` (méthode `url-lib.request.FancyURLopener`), 1121
`prompts`, interpreter, 1549
`propagate` (attribut `logging.Logger`), 615
`property` (classe de base), 18
`property list`, 499
`property_declaration_handler` (dans le module `xml.sax.handler`), 1065
`property_dom_node` (dans le module `xml.sax.handler`), 1065
`property_lexical_handler` (dans le module `xml.sax.handler`), 1065
`property_xml_string` (dans le module `xml.sax.handler`), 1065
`PropertyMock` (classe dans `unittest.mock`), 1424
`prot_c()` (méthode `ftplib.FTP_TLS`), 1143
`prot_p()` (méthode `ftplib.FTP_TLS`), 1143
`proto` (attribut `socket.socket`), 813
`protocol`
 CGI, 1088
 context management, 76
 copy, 403
 FTP, 1121, 1138
 HTTP, 1088, 1121, 1130, 1132, 1181
 IMAP4, 1146
 IMAP4_SSL, 1146
 IMAP4_stream, 1146
 iterator, 34
 NNTP, 1152
 POP3, 1143
 SMTP, 1159
 Telnet, 1168
`protocol` (attribut `ssl.SSLContext`), 837
`Protocol` (classe dans `asyncio`), 886
`PROTOCOL_SSLv2` (dans le module `ssl`), 825
`PROTOCOL_SSLv3` (dans le module `ssl`), 825
`PROTOCOL_SSLv23` (dans le module `ssl`), 825
`PROTOCOL_TLS` (dans le module `ssl`), 824
`PROTOCOL_TLS_CLIENT` (dans le module `ssl`), 824
`PROTOCOL_TLS_SERVER` (dans le module `ssl`), 825
`PROTOCOL_TLSv1` (dans le module `ssl`), 825
`PROTOCOL_TLSv1_1` (dans le module `ssl`), 825
`PROTOCOL_TLSv1_2` (dans le module `ssl`), 825
`protocol_version` (attribut `http.server.BaseHTTPRequestHandler`), 1183
`PROTOCOL_VERSION` (attribut `imaplib.IMAP4`), 1151
`ProtocolError` (classe dans `xmlrpc.client`), 1203
`proxy()` (dans le module `weakref`), 233
`proxyauth()` (méthode `imaplib.IMAP4`), 1149
`ProxyBasicAuthHandler` (classe dans `url-lib.request`), 1108
`ProxyDigestAuthHandler` (classe dans `url-lib.request`), 1108
`ProxyHandler` (classe dans `urllib.request`), 1107
`ProxyType` (dans le module `weakref`), 235
`ProxyTypes` (dans le module `weakref`), 235
`pryear()` (méthode `calendar.TextCalendar`), 200
`ps1` (dans le module `sys`), 1549
`ps2` (dans le module `sys`), 1549
`pstats` (module), 1499
`pstdev()` (dans le module `statistics`), 318
`pthread_kill()` (dans le module `signal`), 923
`pthread_sigmask()` (dans le module `signal`), 924
`pthreads`, 794
`pty`
 module, 529
`pty` (module), 1708
`pu()` (dans le module `turtle`), 1272
`publicId` (attribut `xml.dom.DocumentType`), 1050
`PullDom` (classe dans `xml.dom.pulldom`), 1061
`punctuation` (dans le module `string`), 92
`punctuation_chars` (attribut `shlex.shlex`), 1301
`PurePath` (classe dans `pathlib`), 353
`PurePath.anchor` (dans le module `pathlib`), 357
`PurePath.drive` (dans le module `pathlib`), 356
`PurePath.name` (dans le module `pathlib`), 357
`PurePath.parent` (dans le module `pathlib`), 357
`PurePath.parents` (dans le module `pathlib`), 357
`PurePath.parts` (dans le module `pathlib`), 356
`PurePath.root` (dans le module `pathlib`), 356
`PurePath.stem` (dans le module `pathlib`), 358
`PurePath.suffix` (dans le module `pathlib`), 358

- PurePath.suffixes (*dans le module pathlib*), 358
- PurePosixPath (*classe dans pathlib*), 354
- PureProxy (*classe dans smtpd*), 1166
- PureWindowsPath (*classe dans pathlib*), 354
- purge() (*dans le module re*), 112
- Purpose.CLIENT_AUTH (*dans le module ssl*), 828
- Purpose.SERVER_AUTH (*dans le module ssl*), 828
- push() (*méthode asynchat.async_chat*), 920
- push() (*méthode code.InteractiveConsole*), 1613
- push() (*méthode contextlib.ExitStack*), 1567
- push_alignment() (*méthode formatter.formatter*), 1678
- push_font() (*méthode formatter.formatter*), 1679
- push_margin() (*méthode formatter.formatter*), 1679
- push_source() (*méthode shlex.shlex*), 1300
- push_style() (*méthode formatter.formatter*), 1679
- push_token() (*méthode shlex.shlex*), 1299
- push_with_producer() (*méthode asyn-
chat.async_chat*), 920
- pushbutton() (*méthode msilib.Dialog*), 1688
- put() (*méthode asyncio.Queue*), 908
- put() (*méthode multiprocessing.Queue*), 737
- put() (*méthode multiprocessing.SimpleQueue*), 738
- put() (*méthode queue.Queue*), 792
- put_nowait() (*méthode asyncio.Queue*), 908
- put_nowait() (*méthode multiprocessing.Queue*), 737
- put_nowait() (*méthode queue.Queue*), 792
- putch() (*dans le module msvcrt*), 1690
- putenv() (*dans le module os*), 523
- putheader() (*méthode http.client.HTTPConnection*), 1135
- putp() (*dans le module curses*), 655
- putrequest() (*méthode http.client.HTTPConnection*), 1135
- putwch() (*dans le module msvcrt*), 1690
- putwin() (*méthode curses.window*), 661
- pvariance() (*dans le module statistics*), 318
- pwd
 - module, 369
- pwd (module), 1702
- pwd() (*méthode ftplib.FTP*), 1142
- pwrite() (*dans le module os*), 530
- py_compile (module), 1659
- PY_COMPILED (*dans le module imp*), 1748
- PY_FROZEN (*dans le module imp*), 1748
- py_object (*classe dans ctypes*), 711
- PY_SOURCE (*dans le module imp*), 1748
- pyclbr (module), 1657
- PyCompileError, 1659
- PyDLL (*classe dans ctypes*), 702
- pydoc (module), 1362
- pyexpat
 - module, 1074
- PYFUNCTYPE() (*dans le module ctypes*), 704
- Python 3000, 1762
- Python Editor, 1336
- Python Enhancement Proposals
 - PEP 1, 1762
 - PEP 205, 235
 - PEP 227, 1588
 - PEP 235, 1623
 - PEP 237, 51, 64
 - PEP 238, 1588
 - PEP 249, 419, 420
 - PEP 255, 1588
 - PEP 263, 1623, 1653, 1654
 - PEP 273, 1615
 - PEP 278, 1764
 - PEP 282, 393, 628
 - PEP 292, 100
 - PEP 302, 23, 387, 1548, 1616, 1617, 1621, 1623, 1625, 1628, 1748, 1757, 1759
 - PEP 305, 471
 - PEP 307, 399
 - PEP 324, 774
 - PEP 328, 23, 1588, 1623, 1757
 - PEP 338, 1622
 - PEP 342, 220
 - PEP 343, 1572, 1588, 1755
 - PEP 362, 1599, 1754, 1761
 - PEP 366, 1622, 1623
 - PEP 370, 1608
 - PEP 378, 96
 - PEP 380, 857, 873
 - PEP 383, 154, 798
 - PEP 393, 159, 164, 1548
 - PEP 397, 1526
 - PEP 405, 1523
 - PEP 411, 1546, 1551, 1552, 1762
 - PEP 420, 1623, 1757, 1762
 - PEP 421, 1547, 1760
 - PEP 428, 352
 - PEP 442, 1590
 - PEP 443, 1757
 - PEP 451, 1548, 1618, 1622, 1623, 1757
 - PEP 453, 1522
 - PEP 461, 64
 - PEP 468, 216
 - PEP 475, 18, 88, 529, 531, 532, 557, 576, 807, 809, 811, 848, 852, 855, 925
 - PEP 479, 1588
 - PEP 483, 1347
 - PEP 484, 1347, 1349, 1354, 1360, 1753, 1757, 1764
 - PEP 485, 274, 280
 - PEP 488, 1623, 1634, 1659
 - PEP 489, 1623, 1630, 1633
 - PEP 492, 221, 1546, 1552, 1605, 1754, 1755

PEP 498, 1756
 PEP 506, 515
 PEP 515, 96
 PEP 519, 1761
 PEP 524, 562
 PEP 525, 221, 1546, 1551, 1605, 1754
 PEP 526, 1347, 1359, 1361, 1753, 1764
 PEP 529, 1544, 1552
 PEP 3101, 92
 PEP 3105, 1588
 PEP 3112, 1588
 PEP 3115, 239
 PEP 3116, 1764
 PEP 3118, 65
 PEP 3119, 222, 1575
 PEP 3120, 1623
 PEP 3141, 269, 1575
 PEP 3147, 1622, 1623, 1634, 1659, 1661, 1746, 1747
 PEP 3148, 773
 PEP 3149, 1539
 PEP 3151, 88, 799, 847, 1712
 PEP 3153, 914
 PEP 3154, 399
 PEP 3155, 1762
 PEP 3156, 914
 PEP 3333, 1096, 1098, 1100, 1103
 --python=<interpreter>
 zipapp command line option, 1532
 python_branch() (dans le module platform), 673
 python_build() (dans le module platform), 673
 python_compiler() (dans le module platform), 673
 PYTHON_DOM, 1046
 python_implementation() (dans le module platform), 673
 python_revision() (dans le module platform), 673
 python_version() (dans le module platform), 673
 python_version_tuple() (dans le module platform), 673
 PYTHONASYNCIODEBUG, 866, 909
 PYTHONDOCS, 1363
 PYTHONDONTWRITEBYTECODE, 1541
 PYTHONFAULTHANDLER, 1487
 PYTHONIOENCODING, 1552
 Pythonique, 1762
 PYTHONLEGACYWINDOWSFSENCODING, 1552
 PYTHONNOUSERSITE, 1607
 PYTHONPATH, 1093, 1548
 PYTHONSTARTUP, 142, 1342, 1547, 1607
 PYTHONTRACEMALLOC, 1511, 1515
 PYTHONUSERBASE, 1607
 PyZipFile (classe dans zipfile), 458

Q

-q
 compileall command line option, 1660
 qiflush() (dans le module curses), 655
 QName (classe dans xml.etree.ElementTree), 1042
 qsize() (méthode asyncio.Queue), 908
 qsize() (méthode multiprocessing.Queue), 736
 qsize() (méthode queue.Queue), 791
 quantize() (méthode decimal.Context), 296
 quantize() (méthode decimal.Decimal), 290
 QueryInfoKey() (dans le module winreg), 1693
 QueryReflectionKey() (dans le module winreg), 1695
 QueryValue() (dans le module winreg), 1693
 QueryValueEx() (dans le module winreg), 1694
 queue (attribut sched.scheduler), 790
 Queue (classe dans asyncio), 907
 Queue (classe dans multiprocessing), 736
 Queue (classe dans queue), 791
 queue (module), 791
 Queue() (méthode multiprocessing.managers.SyncManager), 748
 QueueEmpty, 909
 QueueFull, 909
 QueueHandler (classe dans logging.handlers), 648
 QueueListener (classe dans logging.handlers), 649
 quick_ratio() (méthode difflib.SequenceMatcher), 128
 quit (pdb command), 1495
 quit (variable de base), 26
 quit() (méthode ftplib.FTP), 1142
 quit() (méthode nntplib.NNTP), 1154
 quit() (méthode poplib.POP3), 1145
 quit() (méthode smtplib.SMTP), 1164
 quopri (module), 1021
 quote() (dans le module email.utils), 982
 quote() (dans le module shlex), 1298
 quote() (dans le module urllib.parse), 1127
 QUOTE_ALL (dans le module csv), 474
 quote_from_bytes() (dans le module urllib.parse), 1127
 QUOTE_MINIMAL (dans le module csv), 474
 QUOTE_NONE (dans le module csv), 475
 QUOTE_NONNUMERIC (dans le module csv), 474
 quote_plus() (dans le module urllib.parse), 1127
 quoteattr() (dans le module xml.sax.saxutils), 1069
 quotechar (attribut csv.Dialect), 475
 quoted-printable
 encoding, 1021
 quotes (attribut shlex.shlex), 1300
 quoting (attribut csv.Dialect), 475

R

-R

- trace command line option, 1509
- r
 - compileall command line option, 1660
 - trace command line option, 1508
- r N
 - timeit command line option, 1505
- R_OK (dans le module *os*), 534
- radians() (dans le module *math*), 276
- radians() (dans le module *turtle*), 1271
- RadioButtonGroup (classe dans *msilib*), 1688
- radiogroup() (méthode *msilib.Dialog*), 1688
- radix() (méthode *decimal.Context*), 296
- radix() (méthode *decimal.Decimal*), 290
- RADIXCHAR (dans le module *locale*), 1254
- raise
 - état, 81
- raise (2to3 fixer), 1472
- raise_on_defect (attribut *email.policy.Policy*), 948
- RAISE_VARARGS (opcode), 1673
- ramasse-miettes, 1757
- RAND_add() (dans le module *ssl*), 821
- RAND_bytes() (dans le module *ssl*), 821
- RAND_egd() (dans le module *ssl*), 821
- RAND_pseudo_bytes() (dans le module *ssl*), 821
- RAND_status() (dans le module *ssl*), 821
- randbelow() (dans le module *secrets*), 515
- randbits() (dans le module *secrets*), 515
- randint() (dans le module *random*), 310
- random (module), 308
- random() (dans le module *random*), 311
- randrange() (dans le module *random*), 310
- range
 - objet, 40
- range (classe de base), 40
- RARROW (dans le module *token*), 1651
- ratecv() (dans le module *audioop*), 1225
- ratio() (méthode *difflib.SequenceMatcher*), 128
- Rational (classe dans *numbers*), 270
- raw (attribut *io.BufferedIOBase*), 567
- raw() (dans le module *curses*), 655
- raw_data_manager (dans le module *email.contentmanager*), 959
- raw_decode() (méthode *json.JSONDecoder*), 990
- raw_input (2to3 fixer), 1472
- raw_input() (méthode *code.InteractiveConsole*), 1613
- RawArray() (dans le module *multiprocessing.sharedctypes*), 744
- RawConfigParser (classe dans *configparser*), 493
- RawDescriptionHelpFormatter (classe dans *argparse*), 587
- RawIOBase (classe dans *io*), 566
- RawPen (classe dans *turtle*), 1288
- RawTextHelpFormatter (classe dans *argparse*), 587
- RawTurtle (classe dans *turtle*), 1288
- RawValue() (dans le module *multiprocessing.sharedctypes*), 744
- RBRACE (dans le module *token*), 1651
- rcpttos (attribut *smtpd.SMTPChannel*), 1167
- re
 - module, 42, 386
- re (attribut *re.match*), 116
- re (module), 102
- read() (dans le module *os*), 530
- read() (méthode *asyncio.StreamReader*), 893
- read() (méthode *chunk.Chunk*), 1235
- read() (méthode *codecs.StreamReader*), 158
- read() (méthode *configparser.ConfigParser*), 491
- read() (méthode *http.client.HTTPResponse*), 1136
- read() (méthode *imaplib.IMAP4*), 1149
- read() (méthode *io.BufferedIOBase*), 567
- read() (méthode *io.BufferedReader*), 569
- read() (méthode *io.RawIOBase*), 566
- read() (méthode *io.TextIOBase*), 571
- read() (méthode *mimetypes.MimeTypes*), 1014
- read() (méthode *mmap.mmap*), 928
- read() (méthode *ossaudiodev.oss_audio_device*), 1238
- read() (méthode *ssl.MemoryBIO*), 844
- read() (méthode *ssl.SSLSocket*), 829
- read() (méthode *urllib.robotparser.RobotFileParser*), 1129
- read() (méthode *zipfile.ZipFile*), 456
- read1() (méthode *io.BufferedIOBase*), 567
- read1() (méthode *io.BufferedReader*), 569
- read1() (méthode *io.BytesIO*), 569
- read_all() (méthode *telnetlib.Telnet*), 1168
- read_byte() (méthode *mmap.mmap*), 928
- read_bytes() (méthode *pathlib.Path*), 365
- read_dict() (méthode *configparser.ConfigParser*), 492
- read_eager() (méthode *telnetlib.Telnet*), 1169
- read_environ() (dans le module *wsgiref.handlers*), 1103
- read_events() (méthode *xml.etree.ElementTree.XMLPullParser*), 1044
- read_file() (méthode *configparser.ConfigParser*), 491
- read_history_file() (dans le module *readline*), 140
- read_init_file() (dans le module *readline*), 140
- read_lazy() (méthode *telnetlib.Telnet*), 1169
- read_mime_types() (dans le module *mimetypes*), 1013
- read_sb_data() (méthode *telnetlib.Telnet*), 1169
- read_some() (méthode *telnetlib.Telnet*), 1168
- read_string() (méthode *configparser.ConfigParser*), 491
- read_text() (méthode *pathlib.Path*), 365
- read_token() (méthode *shlex.shlex*), 1299
- read_until() (méthode *telnetlib.Telnet*), 1168
- read_very_eager() (méthode *telnetlib.Telnet*), 1168

- `read_very_lazy()` (méthode `telnetlib.Telnet`), 1169
- `read_windows_registry()` (méthode `mimetypes.MimeTypes`), 1015
- `READABLE` (dans le module `tkinter`), 1314
- `readable()` (méthode `asyncore.dispatcher`), 916
- `readable()` (méthode `io.IOBase`), 566
- `readall()` (méthode `io.RawIOBase`), 567
- `reader()` (dans le module `csv`), 472
- `ReadError`, 462
- `readexactly()` (méthode `asyncio.StreamReader`), 893
- `readfp()` (méthode `configparser.ConfigParser`), 493
- `readfp()` (méthode `mimetypes.MimeTypes`), 1014
- `readframes()` (méthode `aifc.aifc`), 1227
- `readframes()` (méthode `sunau.AU_read`), 1230
- `readframes()` (méthode `wave.Wave_read`), 1232
- `readinto()` (méthode `http.client.HTTPResponse`), 1136
- `readinto()` (méthode `io.BufferedIOBase`), 567
- `readinto()` (méthode `io.RawIOBase`), 567
- `readinto1()` (méthode `io.BufferedIOBase`), 568
- `readinto1()` (méthode `io.BytesIO`), 569
- `readline` (module), 139
- `readline()` (méthode `asyncio.StreamReader`), 893
- `readline()` (méthode `codecs.StreamReader`), 158
- `readline()` (méthode `imaplib.IMAP4`), 1150
- `readline()` (méthode `io.IOBase`), 566
- `readline()` (méthode `io.TextIOBase`), 571
- `readline()` (méthode `mmap.mmap`), 928
- `readlines()` (méthode `codecs.StreamReader`), 158
- `readlines()` (méthode `io.IOBase`), 566
- `readlink()` (dans le module `os`), 539
- `readmodule()` (dans le module `pyclbr`), 1657
- `readmodule_ex()` (dans le module `pyclbr`), 1658
- `readonly` (attribut `memoryview`), 69
- `readPlist()` (dans le module `plistlib`), 500
- `readPlistFromBytes()` (dans le module `plistlib`), 500
- `ReadTransport` (classe dans `asyncio`), 884
- `readuntil()` (méthode `asyncio.StreamReader`), 893
- `readv()` (dans le module `os`), 531
- `ready()` (méthode `multiprocessing.pool.AsyncResult`), 754
- `real` (attribut `numbers.Complex`), 269
- `Real` (classe dans `numbers`), 270
- `Real Media File Format`, 1234
- `real_quick_ratio()` (méthode `difflib.SequenceMatcher`), 128
- `realpath()` (dans le module `os.path`), 370
- `reason` (attribut `http.client.HTTPResponse`), 1136
- `reason` (attribut `ssl.SSLError`), 818
- `reason` (attribut `UnicodeError`), 86
- `reason` (attribut `urllib.error.HTTPError`), 1129
- `reason` (attribut `urllib.error.URLError`), 1128
- `reattach()` (méthode `tkinter.ttk.Treeview`), 1326
- `recontrols()` (méthode `ossaudio-dev.oss_mixer_device`), 1241
- `received_data` (attribut `smtpd.SMTPChannel`), 1167
- `received_lines` (attribut `smtpd.SMTPChannel`), 1167
- `recent()` (méthode `imaplib.IMAP4`), 1150
- `records` (attribut `unittest.TestCase`), 1400
- `rect()` (dans le module `cmath`), 278
- `rectangle()` (dans le module `curses.textpad`), 667
- `RecursionError`, 85
- `recursive_repr()` (dans le module `reprlib`), 249
- `recv()` (méthode `asyncore.dispatcher`), 917
- `recv()` (méthode `multiprocessing.connection.Connection`), 740
- `recv()` (méthode `socket.socket`), 809
- `recv_bytes()` (méthode `multiprocessing.connection.Connection`), 740
- `recv_bytes_into()` (méthode `multiprocessing.connection.Connection`), 740
- `recv_into()` (méthode `socket.socket`), 810
- `recvfrom()` (méthode `socket.socket`), 809
- `recvfrom_into()` (méthode `socket.socket`), 810
- `recvmsg()` (méthode `socket.socket`), 809
- `recvmsg_into()` (méthode `socket.socket`), 810
- `redirect_request()` (méthode `url-lib.request.HTTPRedirectHandler`), 1112
- `redirect_stderr()` (dans le module `contextlib`), 1566
- `redirect_stdout()` (dans le module `contextlib`), 1565
- `redisplay()` (dans le module `readline`), 140
- `redrawln()` (méthode `curses.window`), 661
- `redrawwin()` (méthode `curses.window`), 662
- `reduce` (2to3 fixer), 1472
- `reduce()` (dans le module `functools`), 339
- `ref` (classe dans `weakref`), 232
- `ReferenceError`, 85, 235
- `ReferenceType` (dans le module `weakref`), 235
- `refold_source` (attribut `email.policy.EmailPolicy`), 949
- `refresh()` (méthode `curses.window`), 662
- `REG_BINARY` (dans le module `winreg`), 1696
- `REG_DWORD` (dans le module `winreg`), 1696
- `REG_DWORD_BIG_ENDIAN` (dans le module `winreg`), 1697
- `REG_DWORD_LITTLE_ENDIAN` (dans le module `winreg`), 1696
- `REG_EXPAND_SZ` (dans le module `winreg`), 1697
- `REG_FULL_RESOURCE_DESCRIPTOR` (dans le module `winreg`), 1697
- `REG_LINK` (dans le module `winreg`), 1697
- `REG_MULTI_SZ` (dans le module `winreg`), 1697
- `REG_NONE` (dans le module `winreg`), 1697
- `REG_QWORD` (dans le module `winreg`), 1697
- `REG_QWORD_LITTLE_ENDIAN` (dans le module `winreg`), 1697
- `REG_RESOURCE_LIST` (dans le module `winreg`), 1697

`REG_RESOURCE_REQUIREMENTS_LIST` (dans le module `winreg`), 1697

`REG_SZ` (dans le module `winreg`), 1697

`register()` (dans le module `atexit`), 1579

`register()` (dans le module `codecs`), 152

`register()` (dans le module `faulthandler`), 1488

`register()` (dans le module `webbrowser`), 1086

`register()` (méthode `abc.ABCMeta`), 1575

`register()` (méthode `multiprocessing.managers.BaseManager`), 747

`register()` (méthode `select.devpoll`), 849

`register()` (méthode `select.epoll`), 850

`register()` (méthode `selectors.BaseSelector`), 855

`register()` (méthode `select.poll`), 851

`register_adapter()` (dans le module `sqlite3`), 421

`register_archive_format()` (dans le module `shutil`), 394

`register_converter()` (dans le module `sqlite3`), 421

`register_defect()` (méthode `email.policy.Policy`), 948

`register_dialect()` (dans le module `csv`), 472

`register_error()` (dans le module `codecs`), 154

`register_function()` (méthode `xmlrpc.server.CGIXMLRPCRequestHandler`), 1209

`register_function()` (méthode `xmlrpc.server.SimpleXMLRPCServer`), 1206

`register_instance()` (méthode `xmlrpc.server.CGIXMLRPCRequestHandler`), 1209

`register_instance()` (méthode `xmlrpc.server.SimpleXMLRPCServer`), 1206

`register_introspection_functions()` (méthode `xmlrpc.server.CGIXMLRPCRequestHandler`), 1209

`register_introspection_functions()` (méthode `xmlrpc.server.SimpleXMLRPCServer`), 1207

`register_multicall_functions()` (méthode `xmlrpc.server.CGIXMLRPCRequestHandler`), 1209

`register_multicall_functions()` (méthode `xmlrpc.server.SimpleXMLRPCServer`), 1207

`register_namespace()` (dans le module `xml.etree.ElementTree`), 1038

`register_optionflag()` (dans le module `doctest`), 1372

`register_shape()` (dans le module `turtle`), 1286

`register_unpack_format()` (dans le module `shutil`), 394

`registerDOMImplementation()` (dans le module `xml.dom`), 1046

`registerResult()` (dans le module `unittest`), 1413

`relative`
URL, 1122

`relative_to()` (méthode `pathlib.PurePath`), 360

`release()` (dans le module `platform`), 673

`release()` (méthode `_thread.lock`), 795

`release()` (méthode `asyncio.Condition`), 906

`release()` (méthode `asyncio.Lock`), 905

`release()` (méthode `asyncio.Semaphore`), 907

`release()` (méthode `logging.Handler`), 618

`release()` (méthode `memoryview`), 67

`release()` (méthode `multiprocessing.Lock`), 742

`release()` (méthode `multiprocessing.RLock`), 743

`release()` (méthode `threading.Condition`), 722

`release()` (méthode `threading.Lock`), 720

`release()` (méthode `threading.RLock`), 720

`release()` (méthode `threading.Semaphore`), 723

`release_lock()` (dans le module `imp`), 1747

`reload(2to3 fixer)`, 1472

`reload()` (dans le module `imp`), 1746

`reload()` (dans le module `importlib`), 1624

`relpath()` (dans le module `os.path`), 370

`remainder()` (méthode `decimal.Context`), 296

`remainder_near()` (méthode `decimal.Context`), 296

`remainder_near()` (méthode `decimal.Decimal`), 290

`RemoteDisconnected`, 1134

`remove()` (dans le module `os`), 539

`remove()` (méthode `array.array`), 230

`remove()` (méthode `collections.deque`), 208

`remove()` (méthode `frozenset`), 72

`remove()` (méthode `mailbox.Mailbox`), 996

`remove()` (méthode `mailbox.MH`), 1001

`remove()` (méthode `xml.etree.ElementTree.Element`), 1041

`remove()` (sequence method), 37

`remove_done_callback()` (méthode `asyncio.Future`), 876

`remove_flag()` (méthode `mailbox.MaildirMessage`), 1004

`remove_flag()` (méthode `mailbox.mboxMessage`), 1006

`remove_flag()` (méthode `mailbox.MMDFMessage`), 1010

`remove_folder()` (méthode `mailbox.Maildir`), 999

`remove_folder()` (méthode `mailbox.MH`), 1001

`remove_header()` (méthode `urllib.request.Request`), 1110

`remove_history_item()` (dans le module `readline`), 141

`remove_label()` (méthode `mailbox.BabylMessage`), 1008

`remove_option()` (méthode `configparser.ConfigParser`), 492

`remove_option()` (méthode `optparse.OptionParser`), 1736

- `remove_pyc()` (méthode `msilib.Directory`), 1687
`remove_reader()` (méthode `asyncio.AbstractEventLoop`), 863
`remove_section()` (méthode `configparser.ConfigParser`), 492
`remove_sequence()` (méthode `mailbox.MHMessage`), 1007
`remove_signal_handler()` (méthode `asyncio.AbstractEventLoop`), 865
`remove_writer()` (méthode `asyncio.AbstractEventLoop`), 863
`removeAttribute()` (méthode `xml.dom.Element`), 1051
`removeAttributeNode()` (méthode `xml.dom.Element`), 1051
`removeAttributeNS()` (méthode `xml.dom.Element`), 1051
`removeChild()` (méthode `xml.dom.Node`), 1049
`removedirs()` (dans le module `os`), 539
`removeFilter()` (méthode `logging.Handler`), 618
`removeFilter()` (méthode `logging.Logger`), 617
`removeHandler()` (dans le module `unittest`), 1413
`removeHandler()` (méthode `logging.Logger`), 617
`removeResult()` (dans le module `unittest`), 1413
`removexattr()` (dans le module `os`), 550
`rename()` (dans le module `os`), 539
`rename()` (méthode `ftplib.FTP`), 1142
`rename()` (méthode `imaplib.IMAP4`), 1150
`rename()` (méthode `pathlib.Path`), 365
`renames (2to3 fixer)`, 1472
`renames()` (dans le module `os`), 539
`reopenIfNeeded()` (méthode `logging.handlers.WatchedFileHandler`), 640
`reorganize()` (méthode `dbm.gnu.gdbm`), 417
`repeat()` (dans le module `itertools`), 330
`repeat()` (dans le module `timeit`), 1504
`repeat()` (méthode `timeit.Timer`), 1505
`--repeat=N`
 `timeit` command line option, 1505
`repetition`
 operation, 35
`replace()` (dans le module `os`), 540
`replace()` (méthode `bytearray`), 55
`replace()` (méthode `bytes`), 55
`replace()` (méthode `curses.panel.Panel`), 672
`replace()` (méthode `datetime.date`), 175
`replace()` (méthode `datetime.datetime`), 180
`replace()` (méthode `datetime.time`), 186
`replace()` (méthode `inspect.Parameter`), 1598
`replace()` (méthode `inspect.Signature`), 1596
`replace()` (méthode `pathlib.Path`), 365
`replace()` (méthode `str`), 45
`replace_errors()` (dans le module `codecs`), 155
`replace_header()` (méthode `email.message.EmailMessage`), 935
`replace_header()` (méthode `email.message.Message`), 971
`replace_history_item()` (dans le module `readline`), 141
`replace_whitespace` (attribut `textwrap.TextWrapper`), 134
`replaceChild()` (méthode `xml.dom.Node`), 1049
`ReplacePackage()` (dans le module `modulefinder`), 1619
`--report`
 trace command line option, 1508
`report()` (méthode `filecmp.dircmp`), 379
`report()` (méthode `modulefinder.ModuleFinder`), 1620
`REPORT_CDIF` (dans le module `doctest`), 1372
`report_failure()` (méthode `doctest.DocTestRunner`), 1381
`report_full_closure()` (méthode `filecmp.dircmp`), 380
`REPORT_NDIFF` (dans le module `doctest`), 1372
`REPORT_ONLY_FIRST_FAILURE` (dans le module `doctest`), 1372
`report_partial_closure()` (méthode `filecmp.dircmp`), 379
`report_start()` (méthode `doctest.DocTestRunner`), 1381
`report_success()` (méthode `doctest.DocTestRunner`), 1381
`REPORT_UDIFF` (dans le module `doctest`), 1372
`report_unexpected_exception()` (méthode `doctest.DocTestRunner`), 1381
`REPORTING_FLAGS` (dans le module `doctest`), 1372
`repr (2to3 fixer)`, 1472
`Repr` (classe dans `reprlib`), 249
`repr()` (dans le module `reprlib`), 249
`repr()` (fonction de base), 19
`repr()` (méthode `reprlib.Repr`), 250
`repr1()` (méthode `reprlib.Repr`), 250
`reprlib` (module), 249
`Request` (classe dans `urllib.request`), 1106
`request()` (méthode `http.client.HTTPConnection`), 1134
`request_queue_size` (attribut `socketserver.BaseServer`), 1176
`request_rate()` (méthode `urllib.robotparser.RobotFileParser`), 1129
`request_uri()` (dans le module `wsgiref.util`), 1096
`request_version` (attribut `http.server.BaseHTTPRequestHandler`), 1182
`RequestHandlerClass` (attribut `socketserver.BaseServer`), 1176
`requestline` (attribut `http.server.BaseHTTPRequestHandler`), 1182
`requires()` (dans le module `test.support`), 1476

- ul style="list-style-type: none; padding-left: 0;">
- `reserved` (attribut `zipfile.ZipInfo`), 459
- `RESERVED_FUTURE` (dans le module `uuid`), 1172
- `RESERVED_MICROSOFT` (dans le module `uuid`), 1172
- `RESERVED_NCS` (dans le module `uuid`), 1172
- `reset()` (dans le module `turtle`), 1275, 1282
- `reset()` (méthode `bdb.Bdb`), 1484
- `reset()` (méthode `codecs.IncrementalDecoder`), 157
- `reset()` (méthode `codecs.IncrementalEncoder`), 156
- `reset()` (méthode `codecs.StreamReader`), 158
- `reset()` (méthode `codecs.StreamWriter`), 158
- `reset()` (méthode `html.parser.HTMLParser`), 1025
- `reset()` (méthode `ossaudiodev.oss_audio_device`), 1239
- `reset()` (méthode `pipes.Template`), 1712
- `reset()` (méthode `threading.Barrier`), 726
- `reset()` (méthode `xdrllib.Packer`), 496
- `reset()` (méthode `xdrllib.Unpacker`), 497
- `reset()` (méthode `xml.dom.pulldom.DOMEvtStream`), 1062
- `reset()` (méthode `xml.sax.xmlreader.IncrementalParser`), 1072
- `reset_mock()` (méthode `unittest.mock.Mock`), 1418
- `reset_prog_mode()` (dans le module `curses`), 655
- `reset_shell_mode()` (dans le module `curses`), 655
- `resetbuffer()` (méthode `code.InteractiveConsole`), 1613
- `resetlocale()` (dans le module `locale`), 1256
- `resetscreen()` (dans le module `turtle`), 1282
- `resetty()` (dans le module `curses`), 655
- `resetwarnings()` (dans le module `warnings`), 1563
- `resize()` (dans le module `ctypes`), 708
- `resize()` (méthode `curses.window`), 662
- `resize()` (méthode `mmap.mmap`), 928
- `resize_term()` (dans le module `curses`), 655
- `resizemode()` (dans le module `turtle`), 1276
- `resizeterm()` (dans le module `curses`), 655
- `resolution` (attribut `datetime.date`), 174
- `resolution` (attribut `datetime.datetime`), 179
- `resolution` (attribut `datetime.time`), 185
- `resolution` (attribut `datetime.timedelta`), 171
- `resolve()` (méthode `pathlib.Path`), 365
- `resolve_name()` (dans le module `importlib.util`), 1634
- `resolveEntity()` (méthode `xml.sax.handler.EntityResolver`), 1068
- `resource` (module), 1712
- `ResourceDenied`, 1476
- `ResourceLoader` (classe dans `importlib.abc`), 1627
- `ResourceWarning`, 89
- `response` (attribut `nnplib.NNTPError`), 1153
- `response()` (méthode `imaplib.IMAP4`), 1150
- `ResponseNotReady`, 1134
- `responses` (attribut `http.server.BaseHTTPRequestHandler`), 1183
- `responses` (dans le module `http.client`), 1134
- `restart` (`pdb` command), 1495
- `restore()` (dans le module `difflib`), 125
- `restype` (attribut `ctypes._FuncPtr`), 703
- `result()` (méthode `asyncio.Future`), 876
- `result()` (méthode `concurrent.futures.Future`), 772
- `results()` (méthode `trace.Trace`), 1510
- `resume_reading()` (méthode `asyncio.ReadTransport`), 884
- `resume_writing()` (méthode `asyncio.BaseProtocol`), 887
- retours à la ligne universels, 1764
- `retr()` (méthode `poplib.POP3`), 1145
- `retrbinary()` (méthode `ftplib.FTP`), 1141
- `retrieve()` (méthode `urllib.request.URLopener`), 1120
- `retrlines()` (méthode `ftplib.FTP`), 1141
- `return` (`pdb` command), 1493
- `return_annotation` (attribut `inspect.Signature`), 1596
- `return_ok()` (méthode `http.cookiejar.CookiePolicy`), 1193
- `return_value` (attribut `unittest.mock.Mock`), 1420
- `RETURN_VALUE` (opcode), 1669
- `returncode` (attribut `asyncio.subprocess.Process`), 901
- `returncode` (attribut `subprocess.CalledProcessError`), 776
- `returncode` (attribut `subprocess.CompletedProcess`), 775
- `returncode` (attribut `subprocess.Popen`), 782
- `reverse()` (dans le module `audioop`), 1225
- `reverse()` (méthode `array.array`), 231
- `reverse()` (méthode `collections.deque`), 208
- `reverse()` (sequence method), 37
- `reverse_order()` (méthode `pstats.Stats`), 1500
- `reverse_pointer` (attribut `ipaddress.IPv4Address`), 1212
- `reverse_pointer` (attribut `ipaddress.IPv6Address`), 1213
- `reversed()` (fonction de base), 20
- `Reversible` (classe dans `collections.abc`), 220
- `Reversible` (classe dans `typing`), 1355
- `revert()` (méthode `http.cookiejar.FileCookieJar`), 1192
- `rewind()` (méthode `aifc.aifc`), 1227
- `rewind()` (méthode `sunau.AU_read`), 1230
- `rewind()` (méthode `wave.Wave_read`), 1232
- RFC
 - RFC 821, 1159, 1160
 - RFC 822, 577, 960, 977, 1135, 1161, 1163, 1164, 1247
 - RFC 854, 1168
 - RFC 959, 1138, 1141
 - RFC 977, 1152
 - RFC 1014, 496
 - RFC 1123, 577
 - RFC 1321, 503
 - RFC 1422, 838, 846
 - RFC 1521, 1018, 1021

- RFC 1522, 1019, 1021
- RFC 1524, 994, 995
- RFC 1730, 1146
- RFC 1738, 1128
- RFC 1750, 821
- RFC 1766, 1255
- RFC 1808, 1122, 1128
- RFC 1832, 496
- RFC 1869, 1159, 1160
- RFC 1870, 1165, 1167
- RFC 1939, 1143
- RFC 2045, 931, 935, 955, 956, 971, 972, 977, 1015, 1017
- RFC 2045#section-6.8, 1201
- RFC 2046, 931, 959, 977
- RFC 2047, 931, 949, 954, 977, 978, 982
- RFC 2060, 1146, 1151
- RFC 2068, 1186
- RFC 2104, 514
- RFC 2109, 11861188, 1190, 1191, 1195, 1196
- RFC 2183, 931, 937, 973
- RFC 2231, 931, 935, 936, 970972, 977, 984
- RFC 2295, 1132
- RFC 2342, 1149
- RFC 2368, 1128
- RFC 2373, 1212, 1213
- RFC 2396, 1124, 1128
- RFC 2397, 1115
- RFC 2449, 1144
- RFC 2518, 1131
- RFC 2595, 1143, 1145
- RFC 2616, 1097, 1100, 1112, 1113, 1120, 1129
- RFC 2732, 1128
- RFC 2774, 1132
- RFC 2818, 822
- RFC 2821, 931
- RFC 2822, 577, 969, 977, 978, 982, 983, 1003, 1182
- RFC 2964, 1191
- RFC 2965, 1107, 1109, 1190, 1191, 11931197
- RFC 2980, 1152, 1158
- RFC 3056, 1214
- RFC 3171, 1212
- RFC 3229, 1131
- RFC 3280, 829
- RFC 3330, 1213
- RFC 3454, 138
- RFC 3490, 164, 166
- RFC 3490#section-3.1, 166
- RFC 3492, 164, 166
- RFC 3493, 817
- RFC 3501, 1151
- RFC 3542, 806
- RFC 3548, 1015, 1016, 1019
- RFC 3659, 1142
- RFC 3879, 1213
- RFC 3927, 1213
- RFC 3977, 1152, 11541156, 1158
- RFC 3986, 1123, 1125, 1128
- RFC 4086, 846
- RFC 4122, 11701172
- RFC 4180, 471
- RFC 4193, 1213
- RFC 4217, 1139
- RFC 4291, 1213
- RFC 4380, 1214
- RFC 4627, 985, 993
- RFC 4642, 1153
- RFC 4918, 1131, 1132
- RFC 4954, 1162
- RFC 5161, 1148
- RFC 5233, 931, 967
- RFC 5246, 828, 846
- RFC 5280, 822, 846
- RFC 5321, 957, 1165, 1166
- RFC 5322, 932, 933, 941, 944, 945, 947, 949, 950, 952955, 957, 958, 1163
- RFC 5424, 644
- RFC 5735, 1213
- RFC 5842, 1131, 1132
- RFC 5929, 830
- RFC 6066, 827, 835, 846
- RFC 6125, 822
- RFC 6152, 1165
- RFC 6531, 933, 949, 1159, 1165, 1166
- RFC 6532, 931, 932, 941, 949
- RFC 6585, 1132
- RFC 6855, 1148
- RFC 6856, 1145
- RFC 7159, 985, 992, 993
- RFC 7230, 1106, 1136
- RFC 7231, 1131, 1132
- RFC 7232, 1131
- RFC 7233, 1131
- RFC 7235, 1131
- RFC 7238, 1131
- RFC 7301, 827, 834
- RFC 7525, 847
- RFC 7693, 507
- RFC 7914, 506
- rfc2109 (*attribut* `http.cookiejar.Cookie`), 1196
- rfc2109_as_netscape (*attribut* `http.cookiejar.DefaultCookiePolicy`), 1195
- rfc2965 (*attribut* `http.cookiejar.CookiePolicy`), 1194
- RFC_4122 (*dans le module* `uuid`), 1172
- rfile (*attribut* `http.server.BaseHTTPRequestHandler`), 1182
- rfind() (*méthode* `bytearray`), 55

- `rfind()` (méthode bytes), 55
- `rfind()` (méthode `mmap.mmap`), 928
- `rfind()` (méthode str), 46
- `rgb_to_hls()` (dans le module `colorsys`), 1235
- `rgb_to_hsv()` (dans le module `colorsys`), 1235
- `rgb_to_yiq()` (dans le module `colorsys`), 1235
- `rglob()` (méthode `pathlib.Path`), 366
- `right` (attribut `filecmp.dircmp`), 380
- `right()` (dans le module `turtle`), 1266
- `right_list` (attribut `filecmp.dircmp`), 380
- `right_only` (attribut `filecmp.dircmp`), 380
- `RIGHTSHIFT` (dans le module `token`), 1651
- `RIGHTSHIFTEQUAL` (dans le module `token`), 1651
- `rindex()` (méthode `bytearray`), 55
- `rindex()` (méthode bytes), 55
- `rindex()` (méthode str), 46
- `rjust()` (méthode `bytearray`), 56
- `rjust()` (méthode bytes), 56
- `rjust()` (méthode str), 46
- `rlcompleter` (module), 144
- `rlecode_hqx()` (dans le module `binascii`), 1020
- `rledecode_hqx()` (dans le module `binascii`), 1019
- `RLIM_INFINITY` (dans le module `resource`), 1713
- `RLIMIT_AS` (dans le module `resource`), 1714
- `RLIMIT_CORE` (dans le module `resource`), 1713
- `RLIMIT_CPU` (dans le module `resource`), 1713
- `RLIMIT_DATA` (dans le module `resource`), 1713
- `RLIMIT_FSIZE` (dans le module `resource`), 1713
- `RLIMIT_MEMLOCK` (dans le module `resource`), 1714
- `RLIMIT_MSGQUEUE` (dans le module `resource`), 1714
- `RLIMIT_NICE` (dans le module `resource`), 1714
- `RLIMIT_NOFILE` (dans le module `resource`), 1714
- `RLIMIT_NPROC` (dans le module `resource`), 1714
- `RLIMIT_NPTS` (dans le module `resource`), 1715
- `RLIMIT_OFI` (dans le module `resource`), 1714
- `RLIMIT_RSS` (dans le module `resource`), 1714
- `RLIMIT_RTPRIO` (dans le module `resource`), 1714
- `RLIMIT_RTTIME` (dans le module `resource`), 1714
- `RLIMIT_SBSIZE` (dans le module `resource`), 1714
- `RLIMIT_SIGPENDING` (dans le module `resource`), 1714
- `RLIMIT_STACK` (dans le module `resource`), 1714
- `RLIMIT_SWAP` (dans le module `resource`), 1714
- `RLIMIT_VMEM` (dans le module `resource`), 1714
- `RLock` (classe dans `multiprocessing`), 742
- `RLock` (classe dans `threading`), 720
- `RLock()` (méthode `multiprocessing.managers.SyncManager`), 748
- `rmd()` (méthode `ftplib.FTP`), 1142
- `rmdir()` (dans le module `os`), 540
- `rmdir()` (méthode `pathlib.Path`), 366
- `RMFF`, 1234
- `rms()` (dans le module `audioop`), 1225
- `rmtree()` (dans le module `shutil`), 390
- `RobotFileParser` (classe dans `urllib.robotparser`), 1129
- `robots.txt`, 1129
- `rollback()` (méthode `sqlite3.Connection`), 422
- `ROT_THREE` (opcode), 1666
- `ROT_TWO` (opcode), 1666
- `rotate()` (méthode `collections.deque`), 208
- `rotate()` (méthode `decimal.Context`), 296
- `rotate()` (méthode `decimal.Decimal`), 291
- `rotate()` (méthode `logging.handlers.BaseRotatingHandler`), 641
- `RotatingFileHandler` (classe dans `logging.handlers`), 641
- `rotation_filename()` (méthode `logging.handlers.BaseRotatingHandler`), 640
- `rotator` (attribut `logging.handlers.BaseRotatingHandler`), 640
- `round()` (fonction de base), 20
- `ROUND_05UP` (dans le module `decimal`), 298
- `ROUND_CEILING` (dans le module `decimal`), 297
- `ROUND_DOWN` (dans le module `decimal`), 297
- `ROUND_FLOOR` (dans le module `decimal`), 297
- `ROUND_HALF_DOWN` (dans le module `decimal`), 297
- `ROUND_HALF_EVEN` (dans le module `decimal`), 297
- `ROUND_HALF_UP` (dans le module `decimal`), 297
- `ROUND_UP` (dans le module `decimal`), 297
- `Rounded` (classe dans `decimal`), 298
- `Row` (classe dans `sqlite3`), 430
- `row_factory` (attribut `sqlite3.Connection`), 426
- `rowcount` (attribut `sqlite3.Cursor`), 429
- `RPAR` (dans le module `token`), 1651
- `rpartition()` (méthode `bytearray`), 55
- `rpartition()` (méthode bytes), 55
- `rpartition()` (méthode str), 46
- `rpc_paths` (attribut `xmlrpc.server.SimpleXMLRPCRequestHandler`), 1207
- `rpop()` (méthode `poplib.POP3`), 1144
- `rset()` (méthode `poplib.POP3`), 1145
- `rshift()` (dans le module `operator`), 344
- `rsplit()` (méthode `bytearray`), 56
- `rsplit()` (méthode bytes), 56
- `rsplit()` (méthode str), 46
- `RSQB` (dans le module `token`), 1651
- `rstrip()` (méthode `bytearray`), 57
- `rstrip()` (méthode bytes), 57
- `rstrip()` (méthode str), 46
- `rt()` (dans le module `turtle`), 1266
- `RTLD_DEEPBIND` (dans le module `os`), 561
- `RTLD_GLOBAL` (dans le module `os`), 561
- `RTLD_LAZY` (dans le module `os`), 561
- `RTLD_LOCAL` (dans le module `os`), 561
- `RTLD_NODELETE` (dans le module `os`), 561
- `RTLD_NOLOAD` (dans le module `os`), 561
- `RTLD_NOW` (dans le module `os`), 561

ruler (*attribut cmd.Cmd*), 1295
 run (*pdb command*), 1495
 Run script, 1337
 run() (*dans le module pdb*), 1490
 run() (*dans le module profile*), 1498
 run() (*dans le module subprocess*), 774
 run() (*méthode bdb.Bdb*), 1486
 run() (*méthode doctest.DocTestRunner*), 1381
 run() (*méthode multiprocessing.Process*), 733
 run() (*méthode pdb.Pdb*), 1491
 run() (*méthode profile.Profile*), 1498
 run() (*méthode sched.scheduler*), 790
 run() (*méthode threading.Thread*), 718
 run() (*méthode trace.Trace*), 1509
 run() (*méthode unittest.TestCase*), 1396
 run() (*méthode unittest.TestSuite*), 1404
 run() (*méthode unittest.TextTestRunner*), 1409
 run() (*méthode wsgiref.handlers.BaseHandler*), 1101
 run_coroutine_threadsafe() (*dans le module asyncio*), 880
 run_docstring_examples() (*dans le module doc-test*), 1376
 run_doctest() (*dans le module test.support*), 1477
 run_forever() (*méthode asyncio.AbstractEventLoop*), 858
 run_in_executor() (*méthode asyncio.AbstractEventLoop*), 865
 run_module() (*dans le module runpy*), 1621
 run_path() (*dans le module runpy*), 1622
 run_script() (*méthode modulefinder.ModuleFinder*), 1620
 run_unittest() (*dans le module test.support*), 1476
 run_until_complete() (*méthode asyncio.AbstractEventLoop*), 858
 run_with_locale() (*dans le module test.support*), 1478
 runcall() (*dans le module pdb*), 1490
 runcall() (*méthode bdb.Bdb*), 1486
 runcall() (*méthode pdb.Pdb*), 1491
 runcall() (*méthode profile.Profile*), 1499
 runcode() (*méthode code.InteractiveInterpreter*), 1612
 runctx() (*dans le module profile*), 1498
 runctx() (*méthode bdb.Bdb*), 1486
 runctx() (*méthode profile.Profile*), 1499
 runctx() (*méthode trace.Trace*), 1509
 runeval() (*dans le module pdb*), 1490
 runeval() (*méthode bdb.Bdb*), 1486
 runeval() (*méthode pdb.Pdb*), 1491
 runfunc() (*méthode trace.Trace*), 1510
 running() (*méthode concurrent.futures.Future*), 772
 runpy (*module*), 1621
 runsource() (*méthode code.InteractiveInterpreter*), 1612
 RuntimeError, 85

RuntimeWarning, 88
 RUSAGE_BOTH (*dans le module resource*), 1716
 RUSAGE_CHILDREN (*dans le module resource*), 1716
 RUSAGE_SELF (*dans le module resource*), 1716
 RUSAGE_THREAD (*dans le module resource*), 1716

S

-s
 trace command line option, 1509
 unittest-discover command line option, 1389
 S (*dans le module re*), 109
 -s S
 timeit command line option, 1505
 S_ENFMT (*dans le module stat*), 377
 S_IEXEC (*dans le module stat*), 377
 S_IFBLK (*dans le module stat*), 376
 S_IFCHR (*dans le module stat*), 376
 S_IFDIR (*dans le module stat*), 376
 S_IFDOOR (*dans le module stat*), 376
 S_IFIFO (*dans le module stat*), 376
 S_IFLNK (*dans le module stat*), 376
 S_IFMT () (*dans le module stat*), 374
 S_IFPORT (*dans le module stat*), 376
 S_IFREG (*dans le module stat*), 376
 S_IFSOCK (*dans le module stat*), 376
 S_IFWHT (*dans le module stat*), 376
 S_IMODE () (*dans le module stat*), 374
 S_IREAD (*dans le module stat*), 377
 S_IRGRP (*dans le module stat*), 377
 S_IROTH (*dans le module stat*), 377
 S_IRUSR (*dans le module stat*), 377
 S_IRWXG (*dans le module stat*), 377
 S_IRWXO (*dans le module stat*), 377
 S_IRWXU (*dans le module stat*), 377
 S_ISBLK () (*dans le module stat*), 374
 S_ISCHR () (*dans le module stat*), 374
 S_ISDIR () (*dans le module stat*), 374
 S_ISDOOR () (*dans le module stat*), 374
 S_ISFIFO () (*dans le module stat*), 374
 S_ISGID (*dans le module stat*), 377
 S_ISLNK () (*dans le module stat*), 374
 S_ISPORT () (*dans le module stat*), 374
 S_ISREG () (*dans le module stat*), 374
 S_ISSOCK () (*dans le module stat*), 374
 S_ISUID (*dans le module stat*), 377
 S_ISVTX (*dans le module stat*), 377
 S_ISWHT () (*dans le module stat*), 374
 S_IWGRP (*dans le module stat*), 377
 S_IWOTH (*dans le module stat*), 377
 S_IWRITE (*dans le module stat*), 377
 S_IWUSR (*dans le module stat*), 377
 S_IXGRP (*dans le module stat*), 377
 S_IXOTH (*dans le module stat*), 377

- `S_IXUSR` (dans le module `stat`), 377
- `safe_substitute()` (méthode `string.Template`), 100
- `saferepr()` (dans le module `pprint`), 245
- `same_files` (attribut `filecmp.dircmp`), 380
- `same_quantum()` (méthode `decimal.Context`), 296
- `same_quantum()` (méthode `decimal.Decimal`), 291
- `samefile()` (dans le module `os.path`), 370
- `samefile()` (méthode `pathlib.Path`), 366
- `SameFileError`, 389
- `sameopenfile()` (dans le module `os.path`), 371
- `samestat()` (dans le module `os.path`), 371
- `sample()` (dans le module `random`), 310
- `save()` (méthode `http.cookiejar.FileCookieJar`), 1192
- `SaveKey()` (dans le module `winreg`), 1694
- `savetty()` (dans le module `curses`), 655
- `SAX2DOM` (classe dans `xml.dom.pulldom`), 1061
- `SAXException`, 1063
- `SAXNotRecognizedException`, 1063
- `SAXNotSupportedException`, 1063
- `SAXParseException`, 1063
- `scaleb()` (méthode `decimal.Context`), 296
- `scaleb()` (méthode `decimal.Decimal`), 291
- `scandir()` (dans le module `os`), 540
- `scanf()`, 118
- `sched` (module), 789
- `SCHED_BATCH` (dans le module `os`), 558
- `SCHED_FIFO` (dans le module `os`), 558
- `sched_get_priority_max()` (dans le module `os`), 559
- `sched_get_priority_min()` (dans le module `os`), 559
- `sched_getaffinity()` (dans le module `os`), 559
- `sched_getparam()` (dans le module `os`), 559
- `sched_getscheduler()` (dans le module `os`), 559
- `SCHED_IDLE` (dans le module `os`), 558
- `SCHED_OTHER` (dans le module `os`), 558
- `sched_param` (classe dans `os`), 559
- `sched_priority` (attribut `os.sched_param`), 559
- `SCHED_RESET_ON_FORK` (dans le module `os`), 559
- `SCHED_RR` (dans le module `os`), 559
- `sched_rr_get_interval()` (dans le module `os`), 559
- `sched_setaffinity()` (dans le module `os`), 559
- `sched_setparam()` (dans le module `os`), 559
- `sched_setscheduler()` (dans le module `os`), 559
- `SCHED_SPORADIC` (dans le module `os`), 558
- `sched_yield()` (dans le module `os`), 559
- `scheduler` (classe dans `sched`), 789
- `schema` (dans le module `msilib`), 1688
- `Screen` (classe dans `turtle`), 1288
- `screen_size()` (dans le module `turtle`), 1282
- `script_from_examples()` (dans le module `doctest`), 1383
- `scroll()` (méthode `curses.window`), 662
- `ScrolledCanvas` (classe dans `turtle`), 1288
- `scrolllock()` (méthode `curses.window`), 662
- `script()` (dans le module `hashlib`), 506
- `search`
 - path, module, 387, 1548, 1605
- `search()` (dans le module `re`), 110
- `search()` (méthode `imaplib.IMAP4`), 1150
- `search()` (méthode `re.regex`), 113
- `second` (attribut `datetime.datetime`), 179
- `second` (attribut `datetime.time`), 186
- `seconds since the epoch`, 573
- `secrets` (module), 515
- `SECTCRE` (attribut `configparser.ConfigParser`), 488
- `sections()` (méthode `configparser.ConfigParser`), 491
- `secure` (attribut `http.cookiejar.Cookie`), 1196
- `secure hash algorithm`, SHA1, SHA224, SHA256, SHA384, SHA512, 503
- `Secure Sockets Layer`, 817
- `security`
 - CGI, 1092
- `see()` (méthode `tkinter.ttk.Treeview`), 1326
- `seed()` (dans le module `random`), 309
- `seek()` (méthode `chunk.Chunk`), 1235
- `seek()` (méthode `io.IOBase`), 566
- `seek()` (méthode `io.TextIOBase`), 571
- `seek()` (méthode `mmap.mmap`), 929
- `SEEK_CUR` (dans le module `os`), 528
- `SEEK_END` (dans le module `os`), 528
- `SEEK_SET` (dans le module `os`), 528
- `seekable()` (méthode `io.IOBase`), 566
- `seen_greeting` (attribut `smtpd.SMTPChannel`), 1167
- `Select` (classe dans `tkinter.tix`), 1332
- `select` (module), 847
- `select()` (dans le module `select`), 848
- `select()` (méthode `imaplib.IMAP4`), 1150
- `select()` (méthode `selectors.BaseSelector`), 855
- `select()` (méthode `tkinter.ttk.Notebook`), 1321
- `selected_alpn_protocol()` (méthode `ssl.SSLSocket`), 830
- `selected_npn_protocol()` (méthode `ssl.SSLSocket`), 831
- `selection()` (méthode `tkinter.ttk.Treeview`), 1327
- `selection_add()` (méthode `tkinter.ttk.Treeview`), 1327
- `selection_remove()` (méthode `tkinter.ttk.Treeview`), 1327
- `selection_set()` (méthode `tkinter.ttk.Treeview`), 1327
- `selection_toggle()` (méthode `tkinter.ttk.Treeview`), 1327
- `selector` (attribut `urllib.request.Request`), 1109
- `SelectorEventLoop` (classe dans `asyncio`), 870
- `SelectorKey` (classe dans `selectors`), 854
- `selectors` (module), 854

- SelectSelector (classe dans *selectors*), 856
- Semaphore (classe dans *asyncio*), 906
- Semaphore (classe dans *multiprocessing*), 743
- Semaphore (classe dans *threading*), 723
- Semaphore() (méthode *multiprocessing.managers.SyncManager*), 748
- semaphores, binary, 794
- SEMI (dans le module *token*), 1651
- send() (méthode *asyncore.dispatcher*), 917
- send() (méthode *http.client.HTTPConnection*), 1136
- send() (méthode *imaplib.IMAP4*), 1150
- send() (méthode *logging.handlers.DatagramHandler*), 644
- send() (méthode *logging.handlers.SocketHandler*), 643
- send() (méthode *multiprocessing.connection.Connection*), 740
- send() (méthode *socket.socket*), 810
- send_bytes() (méthode *multiprocessing.connection.Connection*), 740
- send_error() (méthode *http.server.BaseHTTPRequestHandler*), 1183
- send_flowing_data() (méthode *formatter.writer*), 1680
- send_header() (méthode *http.server.BaseHTTPRequestHandler*), 1183
- send_hor_rule() (méthode *formatter.writer*), 1680
- send_label_data() (méthode *formatter.writer*), 1680
- send_line_break() (méthode *formatter.writer*), 1680
- send_literal_data() (méthode *formatter.writer*), 1680
- send_message() (méthode *smtplib.SMTP*), 1163
- send_paragraph() (méthode *formatter.writer*), 1680
- send_response() (méthode *http.server.BaseHTTPRequestHandler*), 1183
- send_response_only() (méthode *http.server.BaseHTTPRequestHandler*), 1184
- send_signal() (méthode *asyncio.asyncio.subprocess.Process*), 901
- send_signal() (méthode *asyncio.BaseSubprocessTransport*), 885
- send_signal() (méthode *subprocess.Popen*), 781
- sendall() (méthode *socket.socket*), 811
- sendcmd() (méthode *ftplib.FTP*), 1141
- sendfile() (dans le module *os*), 531
- sendfile() (méthode *socket.socket*), 812
- sendfile() (méthode *wsgiref.handlers.BaseHandler*), 1103
- sendmail() (méthode *smtplib.SMTP*), 1163
- sendmsg() (méthode *socket.socket*), 811
- sendmsg_afalg() (méthode *socket.socket*), 811
- sendto() (méthode *asyncio.DatagramTransport*), 885
- sendto() (méthode *socket.socket*), 811
- sentinel (attribut *multiprocessing.Process*), 734
- sentinel (dans le module *unittest.mock*), 1440
- sep (dans le module *os*), 560
- sequence
- iteration, 34
 - objet, 35
 - types, immutable, 37
 - types, mutable, 37
 - types, operations on, 35, 37
- séquence, 1763
- Sequence (classe dans *collections.abc*), 220
- Sequence (classe dans *typing*), 1355
- sequence (dans le module *msilib*), 1688
- sequence2st() (dans le module *parser*), 1640
- SequenceMatcher (classe dans *difflib*), 122, 126
- serializing
- objects, 397
- serve_forever() (méthode *socketserver.BaseServer*), 1175
- server
- WWW, 1088, 1181
- server (attribut *http.server.BaseHTTPRequestHandler*), 1182
- Server (classe dans *asyncio*), 867
- server_activate() (méthode *socketserver.BaseServer*), 1177
- server_address (attribut *socketserver.BaseServer*), 1176
- server_bind() (méthode *socketserver.BaseServer*), 1177
- server_close() (méthode *socketserver.BaseServer*), 1176
- server_hostname (attribut *ssl.SSLSocket*), 831
- server_side (attribut *ssl.SSLSocket*), 831
- server_software (attribut *wsgiref.handlers.BaseHandler*), 1102
- server_version (attribut *http.server.BaseHTTPRequestHandler*), 1182
- server_version (attribut *http.server.SimpleHTTPRequestHandler*), 1184
- ServerProxy (classe dans *xmlrpc.client*), 1198
- service_actions() (méthode *socketserver.BaseServer*), 1175
- session (attribut *ssl.SSLSocket*), 831
- session_reused (attribut *ssl.SSLSocket*), 831
- session_stats() (méthode *ssl.SSLContext*), 836
- set
- objet, 70
- Set (classe dans *collections.abc*), 220
- Set (classe dans *typing*), 1356
- set (classe de base), 71
- Set Breakpoint, 1339
- set() (méthode *asyncio.Event*), 905
- set() (méthode *configparser.ConfigParser*), 492

- `set()` (méthode `configparser.RawConfigParser`), 493
- `set()` (méthode `http.cookies.Morsel`), 1188
- `set()` (méthode `ossaudiodev.oss_mixer_device`), 1241
- `set()` (méthode `test.support.EnvironmentVarGuard`), 1481
- `set()` (méthode `threading.Event`), 724
- `set()` (méthode `tkinter.ttk.Combobox`), 1319
- `set()` (méthode `tkinter.ttk.Treeview`), 1327
- `set()` (méthode `xml.etree.ElementTree.Element`), 1040
- `SET_ADD` (opcode), 1668
- `set_allowed_domains()` (méthode `http.cookiejar.DefaultCookiePolicy`), 1195
- `set_alpn_protocols()` (méthode `ssl.SSLContext`), 834
- `set_app()` (méthode `wsgiref.simple_server.WSGIServer`), 1099
- `set_asyncgen_hooks()` (dans le module `sys`), 1551
- `set_authorizer()` (méthode `sqlite3.Connection`), 424
- `set_auto_history()` (dans le module `readline`), 141
- `set_blocked_domains()` (méthode `http.cookiejar.DefaultCookiePolicy`), 1195
- `set_blocking()` (dans le module `os`), 531
- `set_boundary()` (méthode `email.message.EmailMessage`), 936
- `set_boundary()` (méthode `email.message.Message`), 972
- `set_break()` (méthode `bdb.Bdb`), 1486
- `set_charset()` (méthode `email.message.Message`), 969
- `set_children()` (méthode `tkinter.ttk.Treeview`), 1325
- `set_ciphers()` (méthode `ssl.SSLContext`), 834
- `set_completer()` (dans le module `readline`), 142
- `set_completer_delims()` (dans le module `readline`), 142
- `set_completion_display_matches_hook()` (dans le module `readline`), 142
- `set_content()` (dans le module `email.contentmanager`), 959
- `set_content()` (méthode `email.contentmanager.ContentManager`), 958
- `set_content()` (méthode `email.message.EmailMessage`), 938
- `set_continue()` (méthode `bdb.Bdb`), 1485
- `set_cookie()` (méthode `http.cookiejar.CookieJar`), 1192
- `set_cookie_if_ok()` (méthode `http.cookiejar.CookieJar`), 1192
- `set_coroutine_wrapper()` (dans le module `sys`), 1551
- `set_current()` (méthode `msilib.Feature`), 1687
- `set_data()` (méthode `importlib.abc.SourceLoader`), 1629
- `set_data()` (méthode `importlib.machinery.SourceFileLoader`), 1632
- `set_date()` (méthode `mailbox.MaildirMessage`), 1004
- `set_debug()` (dans le module `gc`), 1588
- `set_debug()` (méthode `asyncio.AbstractEventLoop`), 866
- `set_debuglevel()` (méthode `ftplib.FTP`), 1140
- `set_debuglevel()` (méthode `http.client.HTTPConnection`), 1135
- `set_debuglevel()` (méthode `nntplib.NNTP`), 1158
- `set_debuglevel()` (méthode `poplib.POP3`), 1144
- `set_debuglevel()` (méthode `smtplib.SMTP`), 1161
- `set_debuglevel()` (méthode `telnetlib.Telnet`), 1169
- `set_default_executor()` (méthode `asyncio.AbstractEventLoop`), 865
- `set_default_type()` (méthode `email.message.EmailMessage`), 936
- `set_default_type()` (méthode `email.message.Message`), 971
- `set_default_verify_paths()` (méthode `ssl.SSLContext`), 834
- `set_defaults()` (méthode `argparse.ArgumentParser`), 609
- `set_defaults()` (méthode `optparse.OptionParser`), 1738
- `set_ecdh_curve()` (méthode `ssl.SSLContext`), 835
- `set_errno()` (dans le module `ctypes`), 708
- `set_event_loop()` (dans le module `asyncio`), 870
- `set_event_loop()` (méthode `asyncio.AbstractEventLoopPolicy`), 872
- `set_event_loop_policy()` (dans le module `asyncio`), 872
- `set_exception()` (méthode `asyncio.Future`), 877
- `set_exception()` (méthode `asyncio.StreamReader`), 893
- `set_exception()` (méthode `concurrent.futures.Future`), 773
- `set_exception_handler()` (méthode `asyncio.AbstractEventLoop`), 866
- `set_executable()` (dans le module `multiprocessing`), 739
- `set_flags()` (méthode `mailbox.MaildirMessage`), 1004
- `set_flags()` (méthode `mailbox.mboxMessage`), 1006
- `set_flags()` (méthode `mailbox.MMDFMessage`), 1010
- `set_from()` (méthode `mailbox.mboxMessage`), 1006
- `set_from()` (méthode `mailbox.MMDFMessage`), 1009
- `set_handle_inheritable()` (dans le module `os`), 533
- `set_history_length()` (dans le module `readline`), 140
- `set_info()` (méthode `mailbox.MaildirMessage`), 1004
- `set_inheritable()` (dans le module `os`), 533
- `set_inheritable()` (méthode `socket.socket`), 812
- `set_labels()` (méthode `mailbox.BabylMessage`), 1008
- `set_last_error()` (dans le module `ctypes`), 708
- `set_literal` (2to3 fixer), 1472
- `set_loader()` (dans le module `importlib.util`), 1635

- `set_next()` (méthode `bdb.Bdb`), 1485
`set_nonstandard_attr()` (méthode `http.cookiejar.Cookie`), 1197
`set_npn_protocols()` (méthode `ssl.SSLContext`), 834
`set_ok()` (méthode `http.cookiejar.CookiePolicy`), 1193
`set_option_negotiation_callback()` (méthode `telnetlib.Telnet`), 1170
`set_output_charset()` (méthode `gettext.NullTranslations`), 1246
`set_package()` (dans le module `importlib.util`), 1635
`set_param()` (méthode `email.message.EmailMessage`), 936
`set_param()` (méthode `email.message.Message`), 972
`set_pasv()` (méthode `ftplib.FTP`), 1141
`set_payload()` (méthode `email.message.Message`), 969
`set_policy()` (méthode `http.cookiejar.CookieJar`), 1192
`set_position()` (méthode `xdrlib.Unpacker`), 497
`set_pre_input_hook()` (dans le module `readline`), 141
`set_progress_handler()` (méthode `sqlite3.Connection`), 424
`set_protocol()` (méthode `asyncio.BaseTransport`), 883
`set_proxy()` (méthode `urllib.request.Request`), 1110
`set_quit()` (méthode `bdb.Bdb`), 1486
`set_recsrc()` (méthode `ossaudio-dev.oss_mixer_device`), 1241
`set_result()` (méthode `asyncio.Future`), 877
`set_result()` (méthode `concurrent.futures.Future`), 772
`set_return()` (méthode `bdb.Bdb`), 1485
`set_running_or_notify_cancel()` (méthode `concurrent.futures.Future`), 772
`set_seq1()` (méthode `difflib.SequenceMatcher`), 127
`set_seq2()` (méthode `difflib.SequenceMatcher`), 127
`set_seqs()` (méthode `difflib.SequenceMatcher`), 127
`set_sequences()` (méthode `mailbox.MH`), 1001
`set_sequences()` (méthode `mailbox.MHMessage`), 1007
`set_server_documentation()` (méthode `xmlrpc.server.DocCGIXMLRPCRequestHandler`), 1210
`set_server_documentation()` (méthode `xmlrpc.server.DocXMLRPCServer`), 1210
`set_server_name()` (méthode `xmlrpc.server.DocCGIXMLRPCRequestHandler`), 1210
`set_server_name()` (méthode `xmlrpc.server.DocXMLRPCServer`), 1210
`set_server_title()` (méthode `xmlrpc.server.DocCGIXMLRPCRequestHandler`), 1210
`set_server_title()` (méthode `xmlrpc.server.DocXMLRPCServer`), 1210
`set_servername_callback()` (méthode `ssl.SSLContext`), 835
`set_spacing()` (méthode `formatter.formatter`), 1679
`set_start_method()` (dans le module `multiprocessing`), 739
`set_startup_hook()` (dans le module `readline`), 141
`set_step()` (méthode `bdb.Bdb`), 1485
`set_subdir()` (méthode `mailbox.MaildirMessage`), 1004
`set_task_factory()` (méthode `asyncio.AbstractEventLoop`), 860
`set_terminator()` (méthode `asyncio.async_chat`), 920
`set_threshold()` (dans le module `gc`), 1589
`set_trace()` (dans le module `bdb`), 1487
`set_trace()` (dans le module `pdb`), 1490
`set_trace()` (méthode `bdb.Bdb`), 1485
`set_trace()` (méthode `pdb.Pdb`), 1491
`set_trace_callback()` (méthode `sqlite3.Connection`), 425
`set_transport()` (méthode `asyncio.StreamReader`), 893
`set_tunnel()` (méthode `http.client.HTTPConnection`), 1135
`set_type()` (méthode `email.message.Message`), 972
`set_unittest_reportflags()` (dans le module `doctest`), 1377
`set_unixfrom()` (méthode `email.message.EmailMessage`), 934
`set_unixfrom()` (méthode `email.message.Message`), 968
`set_until()` (méthode `bdb.Bdb`), 1485
`set_url()` (méthode `url-lib.robotparser.RobotFileParser`), 1129
`set_usage()` (méthode `optparse.OptionParser`), 1738
`set_userptr()` (méthode `curses.panel.Panel`), 672
`set_visible()` (méthode `mailbox.BabylMessage`), 1008
`set_wakeup_fd()` (dans le module `signal`), 924
`set_write_buffer_limits()` (méthode `asyncio.WriteTransport`), 884
`setacl()` (méthode `imaplib.IMAP4`), 1150
`setannotation()` (méthode `imaplib.IMAP4`), 1150
`setattr()` (fonction de base), 20
`setAttribute()` (méthode `xml.dom.Element`), 1052
`setAttributeNode()` (méthode `xml.dom.Element`), 1052
`setAttributeNodeNS()` (méthode `xml.dom.Element`), 1052
`setAttributeNS()` (méthode `xml.dom.Element`), 1052
`SetBase()` (méthode `xml.parsers.expat.xmlparser`), 1075

- setblocking() (méthode *socket.socket*), 812
- setBytesStream() (méthode *xml.sax.xmlreader.InputSource*), 1072
- setcbreak() (dans le module *tty*), 1708
- setCharacterStream() (méthode *xml.sax.xmlreader.InputSource*), 1073
- setcheckinterval() (dans le module *sys*), 1549
- setcomptype() (méthode *aifc.aifc*), 1228
- setcomptype() (méthode *sunau.AU_write*), 1231
- setcomptype() (méthode *wave.Wave_write*), 1233
- setContentHandler() (méthode *xml.sax.xmlreader.XMLReader*), 1071
- setcontext() (dans le module *decimal*), 292
- setDaemon() (méthode *threading.Thread*), 719
- setDefault() (méthode *dict*), 74
- setDefault() (méthode *http.cookies.Morsel*), 1188
- setDefaulttimeout() (dans le module *socket*), 806
- setdlopenflags() (dans le module *sys*), 1550
- setDocumentLocator() (méthode *xml.sax.handler.ContentHandler*), 1066
- setDTDHandler() (méthode *xml.sax.xmlreader.XMLReader*), 1071
- setegid() (dans le module *os*), 524
- setEncoding() (méthode *xml.sax.xmlreader.InputSource*), 1072
- setEntityResolver() (méthode *xml.sax.xmlreader.XMLReader*), 1071
- setErrorHandler() (méthode *xml.sax.xmlreader.XMLReader*), 1071
- seteuid() (dans le module *os*), 524
- setFeature() (méthode *xml.sax.xmlreader.XMLReader*), 1071
- setfirstweekday() (dans le module *calendar*), 200
- setfmt() (méthode *ossaudiodev.oss_audio_device*), 1239
- setFormatter() (méthode *logging.Handler*), 618
- setframerate() (méthode *aifc.aifc*), 1228
- setframerate() (méthode *sunau.AU_write*), 1231
- setframerate() (méthode *wave.Wave_write*), 1233
- setgid() (dans le module *os*), 524
- setgroups() (dans le module *os*), 524
- seth() (dans le module *turtle*), 1267
- setheading() (dans le module *turtle*), 1267
- sethostname() (dans le module *socket*), 806
- SetInteger() (méthode *msilib.Record*), 1686
- setitem() (dans le module *operator*), 345
- setitimer() (dans le module *signal*), 924
- setLevel() (méthode *logging.Handler*), 618
- setLevel() (méthode *logging.Logger*), 615
- setlocale() (dans le module *locale*), 1252
- setLocale() (méthode *xml.sax.xmlreader.XMLReader*), 1071
- setLoggerClass() (dans le module *logging*), 627
- setlogmask() (dans le module *syslog*), 1717
- setLogRecordFactory() (dans le module *logging*), 627
- setmark() (méthode *aifc.aifc*), 1228
- setMaxConns() (méthode *url-lib.request.CacheFTPHandler*), 1116
- setmode() (dans le module *msvcrt*), 1689
- setName() (méthode *threading.Thread*), 718
- setnchannels() (méthode *aifc.aifc*), 1228
- setnchannels() (méthode *sunau.AU_write*), 1231
- setnchannels() (méthode *wave.Wave_write*), 1233
- setnframes() (méthode *aifc.aifc*), 1228
- setnframes() (méthode *sunau.AU_write*), 1231
- setnframes() (méthode *wave.Wave_write*), 1233
- SetParamEntityParsing() (méthode *xml.parsers.expat.xmlparser*), 1075
- setparameters() (méthode *ossaudio-dev.oss_audio_device*), 1240
- setparams() (méthode *aifc.aifc*), 1228
- setparams() (méthode *sunau.AU_write*), 1231
- setparams() (méthode *wave.Wave_write*), 1233
- setpassword() (méthode *zipfile.ZipFile*), 456
- setpgid() (dans le module *os*), 524
- setpgrp() (dans le module *os*), 524
- setpos() (dans le module *turtle*), 1266
- setpos() (méthode *aifc.aifc*), 1227
- setpos() (méthode *sunau.AU_read*), 1230
- setpos() (méthode *wave.Wave_read*), 1233
- setposition() (dans le module *turtle*), 1266
- setpriority() (dans le module *os*), 524
- setprofile() (dans le module *sys*), 1550
- setprofile() (dans le module *threading*), 716
- SetProperty() (méthode *msi-lib.SummaryInformation*), 1685
- setProperty() (méthode *xml.sax.xmlreader.XMLReader*), 1071
- setPublicId() (méthode *xml.sax.xmlreader.InputSource*), 1072
- setquota() (méthode *imaplib.IMAP4*), 1150
- setraw() (dans le module *tty*), 1708
- setrecursionlimit() (dans le module *sys*), 1550
- setregid() (dans le module *os*), 524
- setresgid() (dans le module *os*), 524
- setresuid() (dans le module *os*), 524
- setreuid() (dans le module *os*), 525
- setrlimit() (dans le module *resource*), 1713
- setsampwidth() (méthode *aifc.aifc*), 1228
- setsampwidth() (méthode *sunau.AU_write*), 1231
- setsampwidth() (méthode *wave.Wave_write*), 1233
- setscrreg() (méthode *curses.window*), 662
- setsid() (dans le module *os*), 525
- setsockopt() (méthode *socket.socket*), 812
- setstate() (dans le module *random*), 309
- setstate() (méthode *codecs.IncrementalDecoder*), 157
- setstate() (méthode *codecs.IncrementalEncoder*), 156

- SetStream() (méthode *msilib.Record*), 1686
 SetString() (méthode *msilib.Record*), 1686
 setswitchinterval() (dans le module *sys*), 1550
 setSystemId() (méthode *xml.sax.xmlreader.InputSource*), 1072
 setsyx() (dans le module *curses*), 655
 setTarget() (méthode *ging.handlers.MemoryHandler*), 647
 settiltangle() (dans le module *turtle*), 1277
 settimeout() (méthode *socket.socket*), 812
 setTimeout() (méthode *lib.request.CacheFTPHandler*), 1116
 settrace() (dans le module *sys*), 1550
 settrace() (dans le module *threading*), 716
 setuid() (dans le module *os*), 525
 setundobuffer() (dans le module *turtle*), 1280
 setup() (dans le module *turtle*), 1287
 setup() (méthode *socketserver.BaseRequestHandler*), 1177
 setUp() (méthode *unittest.TestCase*), 1395
 --setup=S
 timeit command line option, 1505
 SETUP_ANNOTATIONS (opcode), 1669
 SETUP_ASYNC_WITH (opcode), 1668
 setup_environ() (méthode *ref.handlers.BaseHandler*), 1102
 SETUP_EXCEPT (opcode), 1672
 SETUP_FINALLY (opcode), 1672
 SETUP_LOOP (opcode), 1672
 setup_python() (méthode *venv.EnvBuilder*), 1527
 setup_scripts() (méthode *venv.EnvBuilder*), 1527
 setup_testing_defaults() (dans le module *wsgi-ref.util*), 1096
 SETUP_WITH (opcode), 1669
 setUpClass() (méthode *unittest.TestCase*), 1395
 setupterm() (dans le module *curses*), 655
 SetValue() (dans le module *winreg*), 1694
 SetValueEx() (dans le module *winreg*), 1694
 setworldcoordinates() (dans le module *turtle*), 1282
 setx() (dans le module *turtle*), 1267
 setxattr() (dans le module *os*), 550
 sety() (dans le module *turtle*), 1267
 SF_APPEND (dans le module *stat*), 378
 SF_ARCHIVED (dans le module *stat*), 378
 SF_IMMUTABLE (dans le module *stat*), 378
 SF_MNOWAIT (dans le module *os*), 531
 SF_NODISKIO (dans le module *os*), 531
 SF_NOUNLINK (dans le module *stat*), 378
 SF_SNAPSHOT (dans le module *stat*), 378
 SF_SYNC (dans le module *os*), 531
 shape (attribut *memoryview*), 70
 Shape (classe dans *turtle*), 1288
 shape() (dans le module *turtle*), 1276
 shapetest() (dans le module *turtle*), 1277
 shapetransform() (dans le module *turtle*), 1278
 share() (méthode *socket.socket*), 812
 shared_ciphers() (méthode *ssl.SSLSocket*), 830
 shearfactor() (dans le module *turtle*), 1277
 Shelf (classe dans *shelve*), 412
 shelve
 module, 413
 shelve (module), 411
 shield() (dans le module *asyncio*), 881
 shift() (méthode *decimal.Context*), 296
 shift() (méthode *decimal.Decimal*), 291
 shift_path_info() (dans le module *wsgiref.util*), 1096
 shifting
 operations, 30
 shlex (classe dans *shlex*), 1299
 shlex (module), 1298
 shortDescription() (méthode *unittest.TestCase*), 1403
 shorten() (dans le module *textwrap*), 132
 shouldFlush() (méthode *ging.handlers.BufferingHandler*), 647
 shouldFlush() (méthode *ging.handlers.MemoryHandler*), 647
 shouldStop (attribut *unittest.TestResult*), 1407
 show() (méthode *curses.panel.Panel*), 672
 show_code() (dans le module *dis*), 1664
 showsyntaxerror() (méthode *code.InteractiveInterpreter*), 1612
 showtraceback() (méthode *code.InteractiveInterpreter*), 1612
 showturtle() (dans le module *turtle*), 1276
 showwarning() (dans le module *warnings*), 1562
 shuffle() (dans le module *random*), 310
 shutdown() (dans le module *logging*), 627
 shutdown() (méthode *concurrent.futures.Executor*), 769
 shutdown() (méthode *imaplib.IMAP4*), 1150
 shutdown() (méthode *multiprocessing.managers.BaseManager*), 747
 shutdown() (méthode *socketserver.BaseServer*), 1176
 shutdown() (méthode *socket.socket*), 812
 shutdown_asyncgens() (méthode *asyncio.AbstractEventLoop*), 858
 shutil (module), 388
 side_effect (attribut *unittest.mock.Mock*), 1420
 SIG_BLOCK (dans le module *signal*), 923
 SIG_DFL (dans le module *signal*), 922
 SIG_IGN (dans le module *signal*), 922
 SIG_SETMASK (dans le module *signal*), 923
 SIG_UNBLOCK (dans le module *signal*), 923
 siginterrupt() (dans le module *signal*), 924
 signal
 module, 795

- ul style="list-style-type: none; padding-left: 0;">
- signal (*module*), 921
- signal() (*dans le module signal*), 924
- signature (*attribut inspect.BoundsArguments*), 1598
- Signature (*classe dans inspect*), 1596
- signature() (*dans le module inspect*), 1595
- sigpending() (*dans le module signal*), 925
- sigtimedwait() (*dans le module signal*), 925
- sigwait() (*dans le module signal*), 925
- sigwaitinfo() (*dans le module signal*), 925
- Simple Mail Transfer Protocol, 1159
- SimpleCookie (*classe dans http.cookies*), 1186
- simplefilter() (*dans le module warnings*), 1563
- SimpleHandler (*classe dans wsgiref.handlers*), 1101
- SimpleHTTPRequestHandler (*classe dans http.server*), 1184
- SimpleNamespace (*classe dans types*), 241
- SimpleQueue (*classe dans multiprocessing*), 738
- SimpleXMLRPCRequestHandler (*classe dans xmlrpc.server*), 1206
- SimpleXMLRPCServer (*classe dans xmlrpc.server*), 1206
- sin() (*dans le module cmath*), 279
- sin() (*dans le module math*), 275
- SingleAddressHeader (*classe dans email.headerregistry*), 955
- singledispatch() (*dans le module functools*), 339
- sinh() (*dans le module cmath*), 279
- sinh() (*dans le module math*), 276
- SIO_KEEPAIVE_VALS (*dans le module socket*), 801
- SIO_LOOPBACK_FAST_PATH (*dans le module socket*), 801
- SIO_RCVALL (*dans le module socket*), 801
- site (*module*), 1605
- site command line option
 - user-base, 1608
 - user-site, 1608
- sitecustomize
 - module, 1606
- site-packages
 - directory, 1605
- sixtofour (*attribut ipaddress.IPv6Address*), 1214
- size (*attribut struct.Struct*), 150
- size (*attribut tarfile.TarInfo*), 466
- size (*attribut tracemalloc.Statistic*), 1518
- size (*attribut tracemalloc.StatisticDiff*), 1518
- size (*attribut tracemalloc.Trace*), 1519
- size() (*méthode ftplib.FTP*), 1142
- size() (*méthode mmap.mmap*), 929
- size_diff (*attribut tracemalloc.StatisticDiff*), 1518
- Sized (*classe dans collections.abc*), 219
- Sized (*classe dans typing*), 1355
- sizeof() (*dans le module ctypes*), 708
- SKIP (*dans le module doctest*), 1371
- skip() (*dans le module unittest*), 1393
- skip() (*méthode chunk.Chunk*), 1235
- skip_unless_symlink() (*dans le module test.support*), 1478
- skipIf() (*dans le module unittest*), 1393
- skipinitialspace (*attribut csv.Dialect*), 475
- skipped (*attribut unittest.TestResult*), 1407
- skippedEntity() (*méthode xml.sax.handler.ContentHandler*), 1067
- SkipTest, 1394
- skipTest() (*méthode unittest.TestCase*), 1396
- skipUnless() (*dans le module unittest*), 1394
- SLASH (*dans le module token*), 1651
- SLASHEQUAL (*dans le module token*), 1651
- slave() (*méthode nntplib.NNTP*), 1158
- sleep() (*dans le module asyncio*), 881
- sleep() (*dans le module time*), 576
- slice
 - assignment, 37
 - fonction de base, 1673
 - operation, 35
- slice (*classe de base*), 20
- SMTP
 - protocol, 1159
- SMTP (*classe dans smtplib*), 1159
- SMTP (*dans le module email.policy*), 951
- smtp_server (*attribut smtpd.SMTPChannel*), 1167
- SMTP_SSL (*classe dans smtplib*), 1159
- smtp_state (*attribut smtpd.SMTPChannel*), 1167
- SMTPAuthenticationError, 1160
- SMTPChannel (*classe dans smtpd*), 1166
- SMTPConnectError, 1160
- smtpd (*module*), 1165
- SMTPDataError, 1160
- SMTPException, 1160
- SMTPHandler (*classe dans logging.handlers*), 646
- SMTPHeloError, 1160
- smtplib (*module*), 1159
- SMTPNotSupportedError, 1160
- SMTPRecipientsRefused, 1160
- SMTPResponseException, 1160
- SMTPSenderRefused, 1160
- SMTPServer (*classe dans smtpd*), 1165
- SMTPServerDisconnected, 1160
- SMTPUTF8 (*dans le module email.policy*), 951
- Snapshot (*classe dans tracemalloc*), 1517
- SNL_ALIAS (*dans le module winsound*), 1698
- SNL_ASYNC (*dans le module winsound*), 1699
- SNL_FILENAME (*dans le module winsound*), 1698
- SNL_LOOP (*dans le module winsound*), 1699
- SNL_MEMORY (*dans le module winsound*), 1699
- SNL_NODEFAULT (*dans le module winsound*), 1699
- SNL_NOSTOP (*dans le module winsound*), 1699
- SNL_NOWAIT (*dans le module winsound*), 1699
- SNL_PURGE (*dans le module winsound*), 1699

- `sndhdr` (*module*), 1237
- `sniff()` (*méthode* `csv.Sniffer`), 474
- `Sniffer` (*classe dans* `csv`), 474
- `sock_accept()` (*méthode* `asyncio.AbstractEventLoop`), 864
- `SOCK_CLOEXEC` (*dans le module* `socket`), 800
- `sock_connect()` (*méthode* `asyncio.AbstractEventLoop`), 864
- `SOCK_DGRAM` (*dans le module* `socket`), 800
- `SOCK_NONBLOCK` (*dans le module* `socket`), 800
- `SOCK_RAW` (*dans le module* `socket`), 800
- `SOCK_RDM` (*dans le module* `socket`), 800
- `sock_recv()` (*méthode* `asyncio.AbstractEventLoop`), 864
- `sock_sendall()` (*méthode* `asyncio.AbstractEventLoop`), 864
- `SOCK_SEQPACKET` (*dans le module* `socket`), 800
- `SOCK_STREAM` (*dans le module* `socket`), 800
- `socket`
 - module*, 1085
 - objet*, 797
- `socket` (*attribut* `socketserver.BaseServer`), 1176
- `socket` (*module*), 797
- `socket()` (*dans le module* `socket`), 802
- `socket()` (*in module* `socket`), 848
- `socket()` (*méthode* `imaplib.IMAP4`), 1150
- `socket_type` (*attribut* `socketserver.BaseServer`), 1176
- `SocketHandler` (*classe dans* `logging.handlers`), 643
- `socketpair()` (*dans le module* `socket`), 802
- `sockets` (*attribut* `asyncio.Server`), 867
- `socketserver` (*module*), 1173
- `SocketType` (*dans le module* `socket`), 803
- `SOL_ALG` (*dans le module* `socket`), 802
- `SOL_RDS` (*dans le module* `socket`), 801
- `SOMAXCONN` (*dans le module* `socket`), 800
- `sort()` (*méthode* `imaplib.IMAP4`), 1150
- `sort()` (*méthode* `list`), 38
- `sort_stats()` (*méthode* `pstats.Stats`), 1499
- `sorted()` (*fonction de base*), 20
- `--sort-keys`
 - `json.tool` *command line option*, 994
- `sortTestMethodsUsing` (*attribut* `unittest.TestLoader`), 1407
- `source` (*attribut* `doctest.Example`), 1379
- `source` (*attribut* `shlex.shlex`), 1301
- `source` (*pdb command*), 1494
- `source_from_cache()` (*dans le module* `imp`), 1747
- `source_from_cache()` (*dans le module* `importlib.util`), 1634
- `SOURCE_SUFFIXES` (*dans le module* `importlib.machinery`), 1630
- `source_to_code()` (*méthode* *statique* `importlib.abc.InspectLoader`), 1628
- `SourceFileLoader` (*classe dans* `importlib.machinery`), 1632
- `sourcehook()` (*méthode* `shlex.shlex`), 1299
- `SourcelessFileLoader` (*classe dans* `importlib.machinery`), 1632
- `SourceLoader` (*classe dans* `importlib.abc`), 1629
- `space`
 - in printf-style formatting*, 50, 63
 - in string formatting*, 95
- `span()` (*méthode* `re.match`), 116
- `spawn()` (*dans le module* `pty`), 1708
- `spawnl()` (*dans le module* `os`), 554
- `spawnle()` (*dans le module* `os`), 554
- `spawnlp()` (*dans le module* `os`), 554
- `spawnlpe()` (*dans le module* `os`), 554
- `spawnv()` (*dans le module* `os`), 554
- `spawnve()` (*dans le module* `os`), 554
- `spawnvp()` (*dans le module* `os`), 554
- `spawnvpe()` (*dans le module* `os`), 554
- `spec_from_file_location()` (*dans le module* `importlib.util`), 1635
- `spec_from_loader()` (*dans le module* `importlib.util`), 1635
- spécificateur de module*, 1760
- `specified_attributes` (*attribut* `xml.parsers.expat.xmlparser`), 1076
- `speed()` (*dans le module* `turtle`), 1269
- `speed()` (*méthode* `ossaudiodev.oss_audio_device`), 1239
- `split()` (*dans le module* `os.path`), 371
- `split()` (*dans le module* `re`), 110
- `split()` (*dans le module* `shlex`), 1298
- `split()` (*méthode* `bytearray`), 57
- `split()` (*méthode* `bytes`), 57
- `split()` (*méthode* `re.regex`), 113
- `split()` (*méthode* `str`), 46
- `splitdrive()` (*dans le module* `os.path`), 371
- `splittext()` (*dans le module* `os.path`), 371
- `splitlines()` (*méthode* `bytearray`), 60
- `splitlines()` (*méthode* `bytes`), 60
- `splitlines()` (*méthode* `str`), 47
- `SplitResult` (*classe dans* `urllib.parse`), 1126
- `SplitResultBytes` (*classe dans* `urllib.parse`), 1126
- `splitunc()` (*dans le module* `os.path`), 371
- `SpooledTemporaryFile()` (*dans le module* `tempfile`), 382
- sprintf-style formatting*, 49, 62
- `spwd` (*module*), 1703
- `sqlite3` (*module*), 419
- `sqlite_version` (*dans le module* `sqlite3`), 420
- `sqlite_version_info` (*dans le module* `sqlite3`), 420
- `sqrt()` (*dans le module* `cmath`), 279
- `sqrt()` (*dans le module* `math`), 275
- `sqrt()` (*méthode* `decimal.Context`), 296
- `sqrt()` (*méthode* `decimal.Decimal`), 291

- SSL, 817
- ssl (module), 817
- SSL_CERT_FILE, 846
- SSL_CERT_PATH, 846
- ssl_version (attribut *fiplib.FTP_TLS*), 1143
- SSLContext (classe dans *ssl*), 832
- SSLError, 818
- SSLErrorNumber (classe dans *ssl*), 828
- SSLObject (classe dans *ssl*), 843
- SSLSession (classe dans *ssl*), 845
- SSLSocket (classe dans *ssl*), 828
- SSLSyscallError, 818
- SSLWantReadError, 818
- SSLWantWriteError, 818
- SSLZeroReturnError, 818
- st () (dans le module *turtle*), 1276
- st2list () (dans le module *parser*), 1641
- st2tuple () (dans le module *parser*), 1641
- st_atime (attribut *os.stat_result*), 543
- ST_ATIME (dans le module *stat*), 376
- st_atime_ns (attribut *os.stat_result*), 543
- st_birthtime (attribut *os.stat_result*), 544
- st_blksize (attribut *os.stat_result*), 544
- st_blocks (attribut *os.stat_result*), 544
- st_creator (attribut *os.stat_result*), 544
- st_ctime (attribut *os.stat_result*), 543
- ST_CTIME (dans le module *stat*), 376
- st_ctime_ns (attribut *os.stat_result*), 543
- st_dev (attribut *os.stat_result*), 543
- ST_DEV (dans le module *stat*), 375
- st_file_attributes (attribut *os.stat_result*), 544
- st_flags (attribut *os.stat_result*), 544
- st_gen (attribut *os.stat_result*), 544
- st_gid (attribut *os.stat_result*), 543
- ST_GID (dans le module *stat*), 375
- st_ino (attribut *os.stat_result*), 543
- ST_INO (dans le module *stat*), 375
- st_mode (attribut *os.stat_result*), 543
- ST_MODE (dans le module *stat*), 375
- st_mtime (attribut *os.stat_result*), 543
- ST_MTIME (dans le module *stat*), 376
- st_mtime_ns (attribut *os.stat_result*), 543
- st_nlink (attribut *os.stat_result*), 543
- ST_NLINK (dans le module *stat*), 375
- st_rdev (attribut *os.stat_result*), 544
- st_rsize (attribut *os.stat_result*), 544
- st_size (attribut *os.stat_result*), 543
- ST_SIZE (dans le module *stat*), 376
- st_type (attribut *os.stat_result*), 544
- st_uid (attribut *os.stat_result*), 543
- ST_UID (dans le module *stat*), 375
- stack (attribut *traceback.TracebackException*), 1583
- stack viewer, 1338
- stack () (dans le module *inspect*), 1602
- stack_effect () (dans le module *dis*), 1665
- stack_size () (dans le module *_thread*), 794
- stack_size () (dans le module *threading*), 716
- stackable
 - streams, 151
- StackSummary (classe dans *traceback*), 1584
- stamp () (dans le module *turtle*), 1268
- standard_b64decode () (dans le module *base64*), 1016
- standard_b64encode () (dans le module *base64*), 1015
- standarderror (2to3 fixer), 1472
- standend () (méthode *curses.window*), 662
- standout () (méthode *curses.window*), 662
- STAR (dans le module *token*), 1651
- STAREQUAL (dans le module *token*), 1651
- starmap () (dans le module *itertools*), 330
- starmap () (méthode *multiprocessing.pool.Pool*), 754
- starmap_async () (méthode *multiprocessing.pool.Pool*), 754
- start (attribut *range*), 40
- start (attribut *UnicodeError*), 86
- start () (dans le module *tracemalloc*), 1515
- start () (méthode *logging.handlers.QueueListener*), 649
- start () (méthode *multiprocessing.managers.BaseManager*), 746
- start () (méthode *multiprocessing.Process*), 733
- start () (méthode *re.match*), 116
- start () (méthode *threading.Thread*), 718
- start () (méthode *tkinter.ttk.Progressbar*), 1322
- start () (méthode *xml.etree.ElementTree.TreeBuilder*), 1043
- start_color () (dans le module *curses*), 655
- start_component () (méthode *msilib.Directory*), 1687
- start_new_thread () (dans le module *_thread*), 794
- start_server () (dans le module *asyncio*), 892
- start_unix_server () (dans le module *asyncio*), 893
- StartCdataSectionHandler () (méthode *xml.parsers.expat.xmlparser*), 1078
- start-directory directory
 - unittest-discover command line option, 1389
- StartDoctypeDeclHandler () (méthode *xml.parsers.expat.xmlparser*), 1077
- startDocument () (méthode *xml.sax.handler.ContentHandler*), 1066
- startElement () (méthode *xml.sax.handler.ContentHandler*), 1066
- StartElementHandler () (méthode *xml.parsers.expat.xmlparser*), 1077
- startElementNS () (méthode *xml.sax.handler.ContentHandler*), 1067

- STARTF_USESHOWWINDOW (dans le module *subprocess*), 783
- STARTF_USESTDHANDLES (dans le module *subprocess*), 783
- startfile() (dans le module *os*), 555
- StartNamespaceDeclHandler() (méthode *xml.parsers.expat.xmlparser*), 1078
- startPrefixMapping() (méthode *xml.sax.handler.ContentHandler*), 1066
- startswith() (méthode *bytearray*), 55
- startswith() (méthode *bytes*), 55
- startswith() (méthode *str*), 47
- startTest() (méthode *unittest.TestResult*), 1408
- startTestRun() (méthode *unittest.TestResult*), 1408
- starttls() (méthode *imaplib.IMAP4*), 1150
- starttls() (méthode *nntplib.NNTP*), 1155
- starttls() (méthode *smtplib.SMTP*), 1162
- STARTUPINFO (classe dans *subprocess*), 782
- stat
- module, 542
- stat (module), 374
- stat() (dans le module *os*), 542
- stat() (méthode *nntplib.NNTP*), 1157
- stat() (méthode *os.DirEntry*), 542
- stat() (méthode *pathlib.Path*), 362
- stat() (méthode *poplib.POP3*), 1145
- stat_float_times() (dans le module *os*), 545
- stat_result (classe dans *os*), 543
- state() (méthode *tkinter.ttk.Widget*), 1318
- staticmethod() (fonction de base), 20
- Statistic (classe dans *tracemalloc*), 1518
- StatisticDiff (classe dans *tracemalloc*), 1518
- statistics (module), 315
- statistics() (méthode *tracemalloc.Snapshot*), 1517
- StatisticsError, 320
- Stats (classe dans *pstats*), 1499
- status (attribut *http.client.HTTPResponse*), 1136
- status() (méthode *imaplib.IMAP4*), 1151
- statvfs() (dans le module *os*), 545
- STD_ERROR_HANDLE (dans le module *subprocess*), 783
- STD_INPUT_HANDLE (dans le module *subprocess*), 783
- STD_OUTPUT_HANDLE (dans le module *subprocess*), 783
- StdButtonBox (classe dans *tkinter.tix*), 1332
- stderr (attribut *asyncio.asyncio.subprocess.Process*), 901
- stderr (attribut *subprocess.CalledProcessError*), 776
- stderr (attribut *subprocess.CompletedProcess*), 775
- stderr (attribut *subprocess.Popen*), 782
- stderr (attribut *subprocess.TimeoutExpired*), 776
- stderr (dans le module *sys*), 1552
- stdev() (dans le module *statistics*), 319
- stdin (attribut *asyncio.asyncio.subprocess.Process*), 901
- stdin (attribut *subprocess.Popen*), 781
- stdin (dans le module *sys*), 1552
- stdout (attribut *asyncio.asyncio.subprocess.Process*), 901
- stdout (attribut *subprocess.CalledProcessError*), 776
- stdout (attribut *subprocess.CompletedProcess*), 775
- stdout (attribut *subprocess.Popen*), 782
- stdout (attribut *subprocess.TimeoutExpired*), 776
- STDOUT (dans le module *subprocess*), 775
- stdout (dans le module *sys*), 1552
- step (attribut *range*), 40
- step (*pdb* command), 1493
- step() (méthode *tkinter.ttk.Progressbar*), 1322
- stereocontrols() (méthode *ossaudio-dev.oss_mixer_device*), 1241
- stls() (méthode *poplib.POP3*), 1145
- stop (attribut *range*), 40
- stop() (dans le module *tracemalloc*), 1516
- stop() (méthode *asyncio.AbstractEventLoop*), 858
- stop() (méthode *logging.handlers.QueueListener*), 649
- stop() (méthode *tkinter.ttk.Progressbar*), 1322
- stop() (méthode *unittest.TestResult*), 1408
- stop_here() (méthode *bdb.Bdb*), 1485
- StopAsyncIteration, 85
- StopIteration, 85
- stopListening() (dans le module *logging.config*), 630
- stopTest() (méthode *unittest.TestResult*), 1408
- stopTestRun() (méthode *unittest.TestResult*), 1408
- storbinary() (méthode *ftplib.FTP*), 1141
- store() (méthode *imaplib.IMAP4*), 1151
- STORE_ACTIONS (attribut *optparse.Option*), 1743
- STORE_ANNOTATION (opcode), 1672
- STORE_ATTR (opcode), 1670
- STORE_DEREF (opcode), 1672
- STORE_FAST (opcode), 1672
- STORE_GLOBAL (opcode), 1670
- STORE_NAME (opcode), 1670
- STORE_SUBSCR (opcode), 1668
- storlines() (méthode *ftplib.FTP*), 1141
- str (built-in class)
- (see also *string*), 41
- str (classe de base), 41
- str() (dans le module *locale*), 1256
- strcoll() (dans le module *locale*), 1256
- StreamError, 462
- StreamHandler (classe dans *logging*), 639
- streamreader (attribut *codecs.CodecInfo*), 151
- StreamReader (classe dans *asyncio*), 893
- StreamReader (classe dans *codecs*), 158
- StreamReaderProtocol (classe dans *asyncio*), 895
- StreamReaderWriter (classe dans *codecs*), 159
- StreamRecoder (classe dans *codecs*), 159
- StreamRequestHandler (classe dans *socketserver*), 1177
- streams, 151
- stackable, 151
- streamwriter (attribut *codecs.CodecInfo*), 151
- StreamWriter (classe dans *asyncio*), 894

- StreamWriter (classe dans *codecs*), 157
- strerror (attribut *OSError*), 84
- strerror() (dans le module *os*), 525
- strftime() (dans le module *time*), 576
- strftime() (méthode *datetime.date*), 176
- strftime() (méthode *datetime.datetime*), 183
- strftime() (méthode *datetime.time*), 187
- strict (attribut *csv.Dialect*), 475
- strict (dans le module *email.policy*), 951
- strict_domain (attribut *http.cookiejar.DefaultCookiePolicy*), 1195
- strict_errors() (dans le module *codecs*), 155
- strict_ns_domain (attribut *http.cookiejar.DefaultCookiePolicy*), 1195
- strict_ns_set_initial_dollar (attribut *http.cookiejar.DefaultCookiePolicy*), 1195
- strict_ns_set_path (attribut *http.cookiejar.DefaultCookiePolicy*), 1195
- strict_ns_unverifiable (attribut *http.cookiejar.DefaultCookiePolicy*), 1195
- strict_rfc2965_unverifiable (attribut *http.cookiejar.DefaultCookiePolicy*), 1195
- strides (attribut *memoryview*), 70
- string
 - format() (built-in function), 11
 - formatting, printf, 49
 - interpolation, printf, 49
 - methods, 42
 - module, 1257
 - objet, 41
 - str (built-in class), 41
 - str() (built-in function), 21
 - text sequence type, 41
- string (attribut *re.match*), 117
- STRING (dans le module *token*), 1651
- string (module), 91
- string_at() (dans le module *ctypes*), 708
- StringIO (classe dans *io*), 572
- stringprep (module), 138
- strip() (méthode *bytearray*), 57
- strip() (méthode *bytes*), 57
- strip() (méthode *str*), 47
- strip_dirs() (méthode *pstats.Stats*), 1499
- stripspaces (attribut *curses.textpad.Textbox*), 668
- strptime() (dans le module *time*), 578
- strptime() (méthode de la classe *datetime.datetime*), 178
- struct
 - module, 812
- Struct (classe dans *struct*), 150
- struct (module), 145
- struct sequence, 1763
- struct_time (classe dans *time*), 578
- Structure (classe dans *ctypes*), 711
- structures
 - C, 145
- strxfrm() (dans le module locale), 1256
- STType (dans le module *parser*), 1642
- Style (classe dans *tkinter.ttk*), 1327
- sub() (dans le module *operator*), 344
- sub() (dans le module *re*), 111
- sub() (méthode *re.regex*), 114
- subdirs (attribut *filecmp.dircmp*), 380
- SubElement() (dans le module *xml.etree.ElementTree*), 1038
- submit() (méthode *concurrent.futures.Executor*), 768
- submodule_search_locations (attribut *importlib.machinery.ModuleSpec*), 1633
- subn() (dans le module *re*), 112
- subn() (méthode *re.regex*), 114
- subnets() (méthode *ipaddress.IPv4Network*), 1217
- subnets() (méthode *ipaddress.IPv6Network*), 1218
- Subnormal (classe dans *decimal*), 299
- suboffsets (attribut *memoryview*), 70
- subpad() (méthode *curses.window*), 662
- subprocess (module), 774
- subprocess_exec() (méthode *asyncio.AbstractEventLoop*), 899
- subprocess_shell() (méthode *asyncio.AbstractEventLoop*), 899
- SubprocessError, 775
- SubprocessProtocol (classe dans *asyncio*), 886
- subscribe() (méthode *imaplib.IMAP4*), 1151
- subscript
 - assignment, 37
 - operation, 35
- subsequent_indent (attribut *textwrap.TextWrapper*), 134
- substitute() (méthode *string.Template*), 100
- subTest() (méthode *unittest.TestCase*), 1396
- subtract() (méthode *collections.Counter*), 205
- subtract() (méthode *decimal.Context*), 297
- subtype (attribut *email.headerregistry.ContentTypeHeader*), 956
- subwin() (méthode *curses.window*), 662
- successful() (méthode *multiprocessing.pool.AsyncResult*), 754
- suffix_map (attribut *mimetypes.MimeTypes*), 1014
- suffix_map (dans le module *mimetypes*), 1013
- suite() (dans le module *parser*), 1640
- suiteClass (attribut *unittest.TestLoader*), 1407
- sum() (fonction de base), 21
- summarize() (méthode *doctest.DocTestRunner*), 1382
- summarize_address_range() (dans le module *ipaddress*), 1221
- summary
 - trace command line option, 1509
- sunau (module), 1229

- `super` (attribut `pycldr.Class`), 1658
 - `super()` (fonction de base), 21
 - `supernet()` (méthode `ipaddress.IPv4Network`), 1217
 - `supernet()` (méthode `ipaddress.IPv6Network`), 1218
 - `supports_bytes_environ` (dans le module `os`), 525
 - `supports_dir_fd` (dans le module `os`), 545
 - `supports_effective_ids` (dans le module `os`), 545
 - `supports_fd` (dans le module `os`), 546
 - `supports_follow_symlinks` (dans le module `os`), 546
 - `supports_unicode_filenames` (dans le module `os.path`), 371
 - `SupportsAbs` (classe dans `typing`), 1355
 - `SupportsBytes` (classe dans `typing`), 1355
 - `SupportsComplex` (classe dans `typing`), 1355
 - `SupportsFloat` (classe dans `typing`), 1355
 - `SupportsInt` (classe dans `typing`), 1355
 - `SupportsRound` (classe dans `typing`), 1355
 - `suppress()` (dans le module `contextlib`), 1565
 - `SuppressCrashReport` (classe dans `test.support`), 1481
 - `SW_HIDE` (dans le module `subprocess`), 783
 - `swapcase()` (méthode `bytearray`), 60
 - `swapcase()` (méthode `bytes`), 60
 - `swapcase()` (méthode `str`), 48
 - `sym_name` (dans le module `symbol`), 1651
 - `Symbol` (classe dans `symtable`), 1650
 - `symbol` (module), 1651
 - `SymbolTable` (classe dans `symtable`), 1649
 - `symlink()` (dans le module `os`), 546
 - `symlink_to()` (méthode `pathlib.Path`), 366
 - `symmetric_difference()` (méthode `frozenset`), 71
 - `symmetric_difference_update()` (méthode `frozenset`), 72
 - `symtable` (module), 1649
 - `symtable()` (dans le module `symtable`), 1649
 - `sync()` (dans le module `os`), 547
 - `sync()` (méthode `dbm.dumb.dumbdbm`), 418
 - `sync()` (méthode `dbm.gnu.gdbm`), 417
 - `sync()` (méthode `ossaudiodev.oss_audio_device`), 1239
 - `sync()` (méthode `shelve.Shelf`), 411
 - `syncdown()` (méthode `curses.window`), 662
 - `synchronized()` (dans le module `multiprocessing.sharedctypes`), 745
 - `SyncManager` (classe dans `multiprocessing.managers`), 747
 - `syncok()` (méthode `curses.window`), 662
 - `syncup()` (méthode `curses.window`), 662
 - `SyntaxErr`, 1054
 - `SyntaxError`, 85
 - `SyntaxWarning`, 88
 - `sys`
 - module, 17
 - `sys` (module), 1539
 - `sys_exc` (2to3 fixer), 1472
 - `sys_version`
 - (attribut `http.server.BaseHTTPRequestHandler`), 1182
 - `sysconf()` (dans le module `os`), 560
 - `sysconf_names` (dans le module `os`), 560
 - `sysconfig` (module), 1554
 - `syslog` (module), 1717
 - `syslog()` (dans le module `syslog`), 1717
 - `SysLogHandler` (classe dans `logging.handlers`), 644
 - `system()` (dans le module `os`), 555
 - `system()` (dans le module `platform`), 673
 - `system_alias()` (dans le module `platform`), 673
 - `SystemError`, 85
 - `SystemExit`, 86
 - `systemId` (attribut `xml.dom.DocumentType`), 1050
 - `SystemRandom` (classe dans `random`), 312
 - `SystemRandom` (classe dans `secrets`), 515
 - `SystemRoot`, 779
- ## T
- T
 - trace command line option, 1509
 - t
 - timeit command line option, 1505
 - trace command line option, 1508
 - unittest-discover command line option, 1389
 - t <tarfile>
 - tarfile command line option, 468
 - t <zipfile>
 - zipfile command line option, 460
 - `T_FMT` (dans le module `locale`), 1254
 - `T_FMT_AMPM` (dans le module `locale`), 1254
 - `tab()` (méthode `tkinter.ttk.Notebook`), 1321
 - `TabError`, 85
 - tableau de correspondances, 1759
 - `tabnanny` (module), 1657
 - `tabs()` (méthode `tkinter.ttk.Notebook`), 1321
 - `tabsize` (attribut `textwrap.TextWrapper`), 134
 - `tabular`
 - data, 471
 - `tag` (attribut `xml.etree.ElementTree.Element`), 1039
 - `tag_bind()` (méthode `tkinter.ttk.Treeview`), 1327
 - `tag_configure()` (méthode `tkinter.ttk.Treeview`), 1327
 - `tag_has()` (méthode `tkinter.ttk.Treeview`), 1327
 - `tagName` (attribut `xml.dom.Element`), 1051
 - `tail` (attribut `xml.etree.ElementTree.Element`), 1039
 - `take_snapshot()` (dans le module `tracemalloc`), 1516
 - `takewhile()` (dans le module `itertools`), 330
 - `tan()` (dans le module `cmath`), 279
 - `tan()` (dans le module `math`), 276
 - `tanh()` (dans le module `cmath`), 279
 - `tanh()` (dans le module `math`), 276

`TarError`, 462
`TarFile` (classe dans `tarfile`), 462, 463
`tarfile` (module), 461
`tarfile` command line option
 `-c <tarfile> <source1> ...`
 `<sourceN>`, 468
 `--create <tarfile> <source1> ...`
 `<sourceN>`, 468
 `-e <tarfile> [<output_dir>]`, 468
 `--extract <tarfile> [<output_dir>]`, 468
 `-l <tarfile>`, 468
 `--list <tarfile>`, 468
 `-t <tarfile>`, 468
 `--test <tarfile>`, 468
 `-v`, 468
 `--verbose`, 468
`target` (attribut `xml.dom.ProcessingInstruction`), 1053
`TarInfo` (classe dans `tarfile`), 466
`Task` (classe dans `asyncio`), 878
`task_done()` (méthode `asyncio.Queue`), 908
`task_done()` (méthode `multiprocessing.JoinableQueue`), 738
`task_done()` (méthode `queue.Queue`), 792
`tau` (dans le module `cmath`), 280
`tau` (dans le module `math`), 277
`tb_locals` (attribut `unittest.TestResult`), 1408
`tbreak` (`pdb` command), 1492
`tcdrain()` (dans le module `termios`), 1707
`tcflow()` (dans le module `termios`), 1707
`tcflush()` (dans le module `termios`), 1707
`tcgetattr()` (dans le module `termios`), 1707
`tcgetpgrp()` (dans le module `os`), 531
`Tcl()` (dans le module `tkinter`), 1304
`TCPServer` (classe dans `socketserver`), 1173
`tcsendbreak()` (dans le module `termios`), 1707
`tcsetattr()` (dans le module `termios`), 1707
`tcsetpgrp()` (dans le module `os`), 532
`tearDown()` (méthode `unittest.TestCase`), 1395
`tearDownClass()` (méthode `unittest.TestCase`), 1396
`tee()` (dans le module `itertools`), 330
`tell()` (méthode `aifc.aifc`), 1227, 1228
`tell()` (méthode `chunk.Chunk`), 1235
`tell()` (méthode `io.IOBase`), 566
`tell()` (méthode `io.TextIOBase`), 571
`tell()` (méthode `mmap.mmap`), 929
`tell()` (méthode `sunau.AU_read`), 1230
`tell()` (méthode `sunau.AU_write`), 1231
`tell()` (méthode `wave.Wave_read`), 1233
`tell()` (méthode `wave.Wave_write`), 1233
`Telnet` (classe dans `telnetlib`), 1168
`telnetlib` (module), 1168
`TEMP`, 383
`temp_cwd()` (dans le module `test.support`), 1478
`temp_dir()` (dans le module `test.support`), 1478
`temp_umask()` (dans le module `test.support`), 1478
`tempdir` (dans le module `tempfile`), 383
`tempfile` (module), 381
`template` (attribut `string.Template`), 101
`Template` (classe dans `pipes`), 1711
`Template` (classe dans `string`), 100
`temporary`
 `file`, 381
 `file name`, 381
`TemporaryDirectory()` (dans le module `tempfile`), 382
`TemporaryFile()` (dans le module `tempfile`), 381
`teredo` (attribut `ipaddress.IPv6Address`), 1214
`TERM`, 655, 656
`termattrs()` (dans le module `curses`), 655
`terminal_size` (classe dans `os`), 532
`terminate()` (méthode `asyncio.subprocess.Process`), 901
`terminate()` (méthode `asyncio.BaseSubprocessTransport`), 885
`terminate()` (méthode `multiprocessing.pool.Pool`), 754
`terminate()` (méthode `multiprocessing.Process`), 734
`terminate()` (méthode `subprocess.Popen`), 781
`termios` (module), 1706
`termname()` (dans le module `curses`), 656
`test` (attribut `doctest.DocTestFailure`), 1384
`test` (attribut `doctest.UnexpectedException`), 1385
`test` (module), 1473
 `--test <tarfile>`
 `tarfile` command line option, 468
`test()` (dans le module `cgi`), 1092
`TestCase` (classe dans `unittest`), 1395
`TestFailed`, 1476
`testfile()` (dans le module `doctest`), 1375
`TESTFN` (dans le module `test.support`), 1476
`TestLoader` (classe dans `unittest`), 1405
`testMethodPrefix` (attribut `unittest.TestLoader`), 1407
`testmod()` (dans le module `doctest`), 1375
`TestResult` (classe dans `unittest`), 1407
`tests` (dans le module `imghdr`), 1236
`testsource()` (dans le module `doctest`), 1384
`testsRun` (attribut `unittest.TestResult`), 1407
`TestSuite` (classe dans `unittest`), 1404
`test.support` (module), 1476
`testzip()` (méthode `zipfile.ZipFile`), 456
`text` (attribut `traceback.TracebackException`), 1583
`text` (attribut `xml.etree.ElementTree.Element`), 1039
`Text` (classe dans `typing`), 1358
`text` (dans le module `msilib`), 1688
`text mode`, 17
`text()` (méthode `msilib.Dialog`), 1688
`text_factory` (attribut `sqlite3.Connection`), 426

- Textbox (*classe dans curses.textpad*), 668
 TextCalendar (*classe dans calendar*), 200
 textdomain() (*dans le module gettext*), 1244
 textdomain() (*dans le module locale*), 1258
 textinput() (*dans le module turtle*), 1285
 TextIO (*classe dans typing*), 1358
 TextIOBase (*classe dans io*), 570
 TextIOWrapper (*classe dans io*), 571
 TextTestResult (*classe dans unittest*), 1409
 TextTestRunner (*classe dans unittest*), 1409
 textwrap (*module*), 132
 TextWrapper (*classe dans textwrap*), 134
 theme_create() (*méthode tkinter.ttk.Style*), 1329
 theme_names() (*méthode tkinter.ttk.Style*), 1330
 theme_settings() (*méthode tkinter.ttk.Style*), 1329
 theme_use() (*méthode tkinter.ttk.Style*), 1330
 THOUSEP (*dans le module locale*), 1254
 Thread (*classe dans threading*), 717
 thread() (*méthode imaplib.IMAP4*), 1151
 thread_info (*dans le module sys*), 1553
 threading (*module*), 715
 ThreadingMixIn (*classe dans socketserver*), 1174
 ThreadingTCPServer (*classe dans socketserver*), 1175
 ThreadingUDPServer (*classe dans socketserver*), 1175
 ThreadPoolExecutor (*classe dans concurrent.futures*), 770
 threads
 POSIX, 794
 throw (2to3 *fixer*), 1472
 ticket_lifetime_hint (*attribut ssl.SSLSession*), 845
 tigetflag() (*dans le module curses*), 656
 tigetnum() (*dans le module curses*), 656
 tigetstr() (*dans le module curses*), 656
 TILDE (*dans le module token*), 1651
 tilt() (*dans le module turtle*), 1277
 tiltangle() (*dans le module turtle*), 1278
 --time
 timeit command line option, 1505
 time (*attribut ssl.SSLSession*), 845
 time (*classe dans datetime*), 185
 time (*module*), 573
 time() (*dans le module time*), 578
 time() (*méthode asyncio.AbstractEventLoop*), 860
 time() (*méthode datetime.datetime*), 180
 Time2Internaldate() (*dans le module imaplib*), 1147
 timedelta (*classe dans datetime*), 171
 TimedRotatingFileHandler (*classe dans logging.handlers*), 642
 timegm() (*dans le module calendar*), 201
 timeit (*module*), 1503
 timeit command line option
 -c, 1506
 --clock, 1506
 -h, 1506
 --help, 1506
 -n N, 1505
 --number=N, 1505
 -p, 1505
 --process, 1505
 -r N, 1505
 --repeat=N, 1505
 -s S, 1505
 --setup=S, 1505
 -t, 1505
 --time, 1505
 -u, 1506
 --unit=U, 1506
 -v, 1506
 --verbose, 1506
 timeit() (*dans le module timeit*), 1504
 timeit() (*méthode timeit.Timer*), 1504
 timeout, 800
 timeout (*attribut socketserver.BaseServer*), 1176
 timeout (*attribut ssl.SSLSession*), 845
 timeout (*attribut subprocess.TimeoutExpired*), 775
 timeout() (*méthode curses.window*), 662
 TIMEOUT_MAX (*dans le module _thread*), 794
 TIMEOUT_MAX (*dans le module threading*), 716
 TimeoutError, 88, 735, 773, 876
 TimeoutExpired, 775
 Timer (*classe dans threading*), 725
 Timer (*classe dans timeit*), 1504
 times() (*dans le module os*), 556
 timestamp() (*méthode datetime.datetime*), 181
 timetuple() (*méthode datetime.date*), 175
 timetuple() (*méthode datetime.datetime*), 181
 timetz() (*méthode datetime.datetime*), 180
 timezone (*classe dans datetime*), 195
 timezone (*dans le module time*), 581
 --timing
 trace command line option, 1509
 title() (*dans le module turtle*), 1287
 title() (*méthode bytearray*), 61
 title() (*méthode bytes*), 61
 title() (*méthode str*), 48
 Tix, 1330
 tix_addbitmapdir() (*méthode tkinter.tix.tixCommand*), 1335
 tix_cget() (*méthode tkinter.tix.tixCommand*), 1334
 tix_configure() (*méthode tkinter.tix.tixCommand*), 1334
 tix_filedialog() (*méthode tkinter.tix.tixCommand*), 1335

- `tix_getbitmap()` (méthode `tkinter.tix.tixCommand`), 1334
- `tix_getimage()` (méthode `tkinter.tix.tixCommand`), 1335
- `TIx_LIBRARY`, 1331
- `tix_option_get()` (méthode `tkinter.tix.tixCommand`), 1335
- `tix_resetoptions()` (méthode `tkinter.tix.tixCommand`), 1335
- `tixCommand` (classe dans `tkinter.tix`), 1334
- `Tk`, 1303
- `Tk` (classe dans `tkinter`), 1304
- `Tk` (classe dans `tkinter.tix`), 1331
- `Tk Option Data Types`, 1311
- `Tkinter`, 1303
- `tkinter` (module), 1303
- `tkinter.scrolledtext` (module), 1335
- `tkinter.tix` (module), 1330
- `tkinter.ttk` (module), 1314
- `TList` (classe dans `tkinter.tix`), 1333
- `TLS`, 817
- `TMP`, 383
- `TMPDIR`, 383
- `to_bytes()` (méthode `int`), 31
- `to_eng_string()` (méthode `decimal.Context`), 297
- `to_eng_string()` (méthode `decimal.Decimal`), 291
- `to_integral()` (méthode `decimal.Decimal`), 291
- `to_integral_exact()` (méthode `decimal.Context`), 297
- `to_integral_exact()` (méthode `decimal.Decimal`), 291
- `to_integral_value()` (méthode `decimal.Decimal`), 291
- `to_sci_string()` (méthode `decimal.Context`), 297
- `ToASCII()` (dans le module `encodings.idna`), 166
- `tobuf()` (méthode `tarfile.TarInfo`), 466
- `tobytes()` (méthode `array.array`), 231
- `tobytes()` (méthode `memoryview`), 66
- `today()` (méthode de la classe `datetime.date`), 173
- `today()` (méthode de la classe `datetime.datetime`), 177
- `tofile()` (méthode `array.array`), 231
- `tok_name` (dans le module `token`), 1651
- `token` (attribut `shlex.shlex`), 1301
- `token` (module), 1651
- `token_bytes()` (dans le module `secrets`), 516
- `token_hex()` (dans le module `secrets`), 516
- `token_urlsafe()` (dans le module `secrets`), 516
- `TokenError`, 1654
- `tokenize` (module), 1653
- `tokenize` command line option
 - `-e`, 1655
 - `--exact`, 1655
 - `-h`, 1655
 - `--help`, 1655
- `tokenize()` (dans le module `tokenize`), 1653
- `tolist()` (méthode `array.array`), 231
- `tolist()` (méthode `memoryview`), 66
- `tolist()` (méthode `parser.ST`), 1642
- `tomono()` (dans le module `audioop`), 1225
- `toordinal()` (méthode `datetime.date`), 175
- `toordinal()` (méthode `datetime.datetime`), 181
- `top()` (méthode `curses.panel.Panel`), 672
- `top()` (méthode `poplib.POP3`), 1145
- `top_panel()` (dans le module `curses.panel`), 671
- `--top-level-directory` directory
 - `unittest-discover` command line option, 1389
- `toprettyxml()` (méthode `xml.dom.minidom.Node`), 1058
- `tostereo()` (dans le module `audioop`), 1225
- `tostring()` (dans le module `xml.etree.ElementTree`), 1038
- `tostring()` (méthode `array.array`), 231
- `tostringlist()` (dans le module `xml.etree.ElementTree`), 1039
- `total_changes` (attribut `sqlite3.Connection`), 427
- `total_ordering()` (dans le module `functools`), 337
- `total_seconds()` (méthode `datetime.timedelta`), 173
- `totuple()` (méthode `parser.ST`), 1642
- `touch()` (méthode `pathlib.Path`), 366
- `touchline()` (méthode `curses.window`), 663
- `touchwin()` (méthode `curses.window`), 663
- `ToUnicode()` (dans le module `encodings.idna`), 166
- `tounicode()` (méthode `array.array`), 231
- `towards()` (dans le module `turtle`), 1270
- `toxml()` (méthode `xml.dom.minidom.Node`), 1058
- `tparm()` (dans le module `curses`), 656
- `--trace`
 - `trace` command line option, 1508
- `Trace` (classe dans `trace`), 1509
- `Trace` (classe dans `tracemalloc`), 1519
- `trace` (module), 1508
- `trace` command line option
 - `-C`, 1509
 - `-c`, 1508
 - `--count`, 1508
 - `--coverdir=<dir>`, 1509
 - `-f`, 1509
 - `--file=<file>`, 1509
 - `-g`, 1509
 - `--help`, 1508
 - `--ignore-dir=<dir>`, 1509
 - `--ignore-module=<mod>`, 1509
 - `-l`, 1508
 - `--listfuncs`, 1508
 - `-m`, 1509
 - `--missing`, 1509
 - `--no-report`, 1509

- R, 1509
- r, 1508
- report, 1508
- s, 1509
- summary, 1509
- T, 1509
- t, 1508
- timing, 1509
- trace, 1508
- trackcalls, 1509
- version, 1508
- trace function, 716, 1545, 1550
- trace() (dans le module inspect), 1602
- trace_dispatch() (méthode bdb.Bdb), 1484
- traceback
 - objet, 1541, 1581
- traceback (attribut tracemalloc.Statistic), 1518
- traceback (attribut tracemalloc.StatisticDiff), 1518
- traceback (attribut tracemalloc.Trace), 1519
- Traceback (classe dans tracemalloc), 1519
- traceback (module), 1581
- traceback_limit (attribut tracemalloc.Snapshot), 1518
- traceback_limit (attribut wsgi-ref.handlers.BaseHandler), 1103
- TracebackException (classe dans traceback), 1583
- tracebacklimit (dans le module sys), 1553
- tracebacks
 - in CGI scripts, 1095
- TracebackType (dans le module types), 240
- tracemalloc (module), 1510
- tracer() (dans le module turtle), 1283
- traces (attribut tracemalloc.Snapshot), 1518
- trackcalls
 - trace command line option, 1509
- tranche, 1763
- transfercmd() (méthode ftplib.FTP), 1141
- TransientResource (classe dans test.support), 1480
- translate() (dans le module fnmatch), 387
- translate() (méthode bytearray), 55
- translate() (méthode bytes), 55
- translate() (méthode str), 48
- translation() (dans le module gettext), 1245
- transport (attribut asyncio.StreamWriter), 894
- Transport Layer Security, 817
- Tree (classe dans tkinter.tix), 1333
- TreeBuilder (classe dans xml.etree.ElementTree), 1043
- Treeview (classe dans tkinter.ttk), 1325
- triangular() (dans le module random), 311
- True, 27, 79
- true, 27
- True (variable de base), 25
- truediv() (dans le module operator), 344
- trunc() (dans le module math), 274
- trunc() (in module math), 30
- truncate() (dans le module os), 547
- truncate() (méthode io.IOBase), 566
- truth
 - value, 27
- truth() (dans le module operator), 343
- try
 - état, 81
- ttk, 1314
- tty
 - I/O control, 1706
- tty (module), 1708
- ttyname() (dans le module os), 532
- tuple
 - objet, 37, 39
- tuple (classe de base), 39
- Tuple (dans le module typing), 1361
- tuple2st() (dans le module parser), 1641
- tuple_params (2to3 fixer), 1472
- turnoff_sigfpe() (dans le module fpectl), 1608
- turnon_sigfpe() (dans le module fpectl), 1608
- Turtle (classe dans turtle), 1288
- turtle (module), 1261
- turtledemo (module), 1291
- turtles() (dans le module turtle), 1286
- TurtleScreen (classe dans turtle), 1288
- turtlesize() (dans le module turtle), 1277
- type, 1764
 - Boolean, 6
 - fonction de base, 78
 - objet, 22
 - operations on dictionary, 73
 - operations on list, 37
- type (attribut optparse.Option), 1732
- type (attribut socket.socket), 813
- type (attribut tarfile.TarInfo), 466
- type (attribut urllib.request.Request), 1109
- Type (classe dans typing), 1354
- type (classe de base), 22
- TYPE_CHECKER (attribut optparse.Option), 1742
- TYPE_CHECKING (dans le module typing), 1362
- typeahead() (dans le module curses), 656
- typecode (attribut array.array), 229
- typecodes (dans le module array), 229
- TYPED_ACTIONS (attribut optparse.Option), 1744
- typed_subpart_iterator() (dans le module email.iterators), 984
- TypeError, 86
- types
 - built-in, 27
 - immutable sequence, 37
 - module, 78
 - mutable sequence, 37
 - operations on integer, 30

- operations on mapping, 73
- operations on numeric, 29
- operations on sequence, 35, 37
- types (2to3 fixer), 1472
- TYPES (attribut *optparse.Option*), 1742
- types (module), 239
- types_map (attribut *mimetypes.MimeTypes*), 1014
- types_map (dans le module *mimetypes*), 1013
- types_map_inv (attribut *mimetypes.MimeTypes*), 1014
- TypeVar (classe dans *typing*), 1353
- typing (module), 1347
- TZ, 579, 580
- tzinfo (attribut *datetime.datetime*), 179
- tzinfo (attribut *datetime.time*), 186
- tzinfo (classe dans *datetime*), 188
- tzname (dans le module *time*), 581
- tzname() (méthode *datetime.datetime*), 181
- tzname() (méthode *datetime.time*), 187
- tzname() (méthode *datetime.timezone*), 195
- tzname() (méthode *datetime.tzinfo*), 189
- tzset() (dans le module *time*), 579

U

-u

- timeit command line option, 1506
- ucd_3_2_0 (dans le module *unicodedata*), 137
- udata (attribut *select.kevent*), 853
- UDPServer (classe dans *socketserver*), 1173
- UF_APPEND (dans le module *stat*), 378
- UF_COMPRESSED (dans le module *stat*), 378
- UF_HIDDEN (dans le module *stat*), 378
- UF_IMMUTABLE (dans le module *stat*), 378
- UF_NODUMP (dans le module *stat*), 378
- UF_NOUNLINK (dans le module *stat*), 378
- UF_OPAQUE (dans le module *stat*), 378
- uid (attribut *tarfile.TarInfo*), 466
- uid() (méthode *imaplib.IMAP4*), 1151
- uidl() (méthode *poplib.POP3*), 1145
- u-LAW, 1223, 1228, 1237
- ulaw2lin() (dans le module *audioop*), 1225
- umask() (dans le module *os*), 525
- unalias (*pdb* command), 1495
- uname (attribut *tarfile.TarInfo*), 466
- uname() (dans le module *os*), 525
- uname() (dans le module *platform*), 674
- UNARY_INVERT (opcode), 1666
- UNARY_NEGATIVE (opcode), 1666
- UNARY_NOT (opcode), 1666
- UNARY_POSITIVE (opcode), 1666
- UnboundLocalError, 86
- unbuffered I/O, 17
- UNC paths
 - and *os.makedirs()*, 537
- unconsumed_tail (attribut *zlib.Decompress*), 442

- unctrl() (dans le module *curses*), 656
- unctrl() (dans le module *curses.ascii*), 670
- Underflow (classe dans *decimal*), 299
- undisplay (*pdb* command), 1494
- undo() (dans le module *turtle*), 1269
- undobufferentries() (dans le module *turtle*), 1280
- undoc_header (attribut *cmd.Cmd*), 1295
- unescape() (dans le module *html*), 1023
- unescape() (dans le module *xml.sax.saxutils*), 1069
- UnexpectedException, 1385
- unexpectedSuccesses (attribut *unittest.TestResult*), 1407
- unget_wch() (dans le module *curses*), 656
- ungetch() (dans le module *curses*), 656
- ungetch() (dans le module *msvcrt*), 1690
- ungetmouse() (dans le module *curses*), 656
- ungetwch() (dans le module *msvcrt*), 1690
- unhexlify() (dans le module *binascii*), 1020
- Unicode, 136, 151
 - database, 136
- unicode (2to3 fixer), 1473
- unicodedata (module), 136
- UnicodeDecodeError, 86
- UnicodeEncodeError, 86
- UnicodeError, 86
- UnicodeTranslateError, 86
- UnicodeWarning, 89
- unidata_version (dans le module *unicodedata*), 137
- unified_diff() (dans le module *difflib*), 125
- uniform() (dans le module *random*), 311
- UnimplementedFileMode, 1133
- Union (classe dans *ctypes*), 711
- Union (dans le module *typing*), 1360
- union() (méthode *frozenset*), 71
- unique() (dans le module *enum*), 251, 254
- unit=U
 - timeit command line option, 1506
- unittest (module), 1386
- unittest command line option
 - b, 1389
 - buffer, 1389
 - c, 1389
 - catch, 1389
 - f, 1389
 - failfast, 1389
 - locals, 1389
- unittest-discover command line option
 - p, 1389
 - pattern pattern, 1389
 - s, 1389
 - start-directory directory, 1389
 - t, 1389
 - top-level-directory directory, 1389

- v, 1389
- verbose, 1389
- unittest.mock (*module*), 1413
- universal newlines
 - bytearray.splitlines method, 60
 - bytes.splitlines method, 60
 - csv.reader function, 472
 - importlib.abc.InspectLoader.get_source method, 1628
 - io.IncrementalNewlineDecoder class, 572
 - io.TextIOWrapper class, 571
 - open() built-in function, 17
 - str.splitlines method, 47
 - subprocess module, 776
- UNIX
 - file control, 1709
 - I/O control, 1709
- unix_dialect (*classe dans csv*), 474
- UnixDatagramServer (*classe dans socketserver*), 1173
- UnixStreamServer (*classe dans socketserver*), 1173
- unknown_decl() (*méthode html.parser.HTMLParser*), 1026
- unknown_open() (*méthode lib.request.BaseHandler*), 1112
- unknown_open() (*méthode lib.request.UnknownHandler*), 1116
- UnknownHandler (*classe dans urllib.request*), 1109
- UnknownProtocol, 1133
- UnknownTransferEncoding, 1133
- unlink() (*dans le module os*), 547
- unlink() (*méthode pathlib.Path*), 366
- unlink() (*méthode xml.dom.minidom.Node*), 1057
- unlock() (*méthode mailbox.Babyl*), 1002
- unlock() (*méthode mailbox.Mailbox*), 998
- unlock() (*méthode mailbox.Maildir*), 999
- unlock() (*méthode mailbox.mbox*), 1000
- unlock() (*méthode mailbox.MH*), 1001
- unlock() (*méthode mailbox.MMDf*), 1003
- unpack() (*dans le module struct*), 146
- unpack() (*méthode struct.Struct*), 150
- unpack_archive() (*dans le module shutil*), 394
- unpack_array() (*méthode xdrlib.Unpacker*), 498
- unpack_bytes() (*méthode xdrlib.Unpacker*), 498
- unpack_double() (*méthode xdrlib.Unpacker*), 497
- UNPACK_EX (*opcode*), 1670
- unpack_farray() (*méthode xdrlib.Unpacker*), 498
- unpack_float() (*méthode xdrlib.Unpacker*), 497
- unpack_fopaque() (*méthode xdrlib.Unpacker*), 498
- unpack_from() (*dans le module struct*), 146
- unpack_from() (*méthode struct.Struct*), 150
- unpack_fstring() (*méthode xdrlib.Unpacker*), 497
- unpack_list() (*méthode xdrlib.Unpacker*), 498
- unpack_opaque() (*méthode xdrlib.Unpacker*), 498
- UNPACK_SEQUENCE (*opcode*), 1670
- unpack_string() (*méthode xdrlib.Unpacker*), 498
- Unpacker (*classe dans xdrlib*), 496
- unparsedEntityDecl() (*méthode xml.sax.handler.DTDHandler*), 1068
- UnparsedEntityDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1077
- Unpickler (*classe dans pickle*), 401
- UnpicklingError, 400
- unquote() (*dans le module email.utils*), 982
- unquote() (*dans le module urllib.parse*), 1127
- unquote_plus() (*dans le module urllib.parse*), 1127
- unquote_to_bytes() (*dans le module urllib.parse*), 1127
- unregister() (*dans le module atexit*), 1580
- unregister() (*dans le module faulthandler*), 1488
- unregister() (*méthode select.devpoll*), 849
- unregister() (*méthode select.epoll*), 850
- unregister() (*méthode selectors.BaseSelector*), 855
- unregister() (*méthode select.poll*), 851
- unregister_archive_format() (*dans le module shutil*), 394
- unregister_dialect() (*dans le module csv*), 473
- unregister_unpack_format() (*dans le module shutil*), 394
- unset() (*méthode test.support.EnvironmentVarGuard*), 1481
- unsetenv() (*dans le module os*), 525
- UnstructuredHeader (*classe dans email.headerregistry*), 954
- unsubscribe() (*méthode imaplib.IMAP4*), 1151
- UnsupportedOperation, 564
- until (*pdb command*), 1493
- untokenize() (*dans le module tokenize*), 1654
- untouchwin() (*méthode curses.window*), 663
- unused_data (attribut bz2.BZ2Decompressor), 447
- unused_data (attribut lzma.LZMADecompressor), 451
- unused_data (attribut zlib.Decompress), 441
- unverifiable (attribut urllib.request.Request), 1109
- unwrap() (*dans le module inspect*), 1601
- unwrap() (*méthode ssl.SSLSocket*), 831
- up (*pdb command*), 1492
- up() (*dans le module turtle*), 1272
- update() (*dans le module turtle*), 1283
- update() (*méthode collections.Counter*), 206
- update() (*méthode dict*), 74
- update() (*méthode frozenset*), 72
- update() (*méthode hashlib.hash*), 505
- update() (*méthode hmac.HMAC*), 514
- update() (*méthode http.cookies.Morsel*), 1188
- update() (*méthode mailbox.Mailbox*), 998
- update() (*méthode mailbox.Maildir*), 999
- update() (*méthode trace.CoverageResults*), 1510

- ul>
- `update_authenticated()` (méthode `urllib.request.HTTPPasswordMgrWithPriorAuth`), 1114
- `update_lines_cols()` (dans le module `curses`), 656
- `update_panels()` (dans le module `curses.panel`), 671
- `update_visible()` (méthode `mailbox.BabylMessage`), 1008
- `update_wrapper()` (dans le module `functools`), 341
- `upper()` (méthode `bytearray`), 61
- `upper()` (méthode `bytes`), 61
- `upper()` (méthode `str`), 49
- `urandom()` (dans le module `os`), 561
- `URL`, 1088, 1122, 1129, 1181
 - `parsing`, 1122
 - `relative`, 1122
- `url` (attribut `xmlrpc.client.ProtocolError`), 1203
- `url2pathname()` (dans le module `urllib.request`), 1106
- `urlcleanup()` (dans le module `urllib.request`), 1119
- `urldefrag()` (dans le module `urllib.parse`), 1125
- `urlencode()` (dans le module `urllib.parse`), 1127
- `URLError`, 1128
- `urljoin()` (dans le module `urllib.parse`), 1124
- `urllib` (2to3 fixer), 1473
- `urllib` (module), 1104
- `urllib.error` (module), 1128
- `urllib.parse` (module), 1122
- `urllib.request`
 - module, 1132
- `urllib.request` (module), 1104
- `urllib.response` (module), 1121
- `urllib.robotparser` (module), 1129
- `urlopen()` (dans le module `urllib.request`), 1104
- `URLOpener` (classe dans `urllib.request`), 1119
- `urlparse()` (dans le module `urllib.parse`), 1122
- `urlretrieve()` (dans le module `urllib.request`), 1119
- `urlsafe_b64decode()` (dans le module `base64`), 1016
- `urlsafe_b64encode()` (dans le module `base64`), 1016
- `urlsplit()` (dans le module `urllib.parse`), 1124
- `urlunparse()` (dans le module `urllib.parse`), 1124
- `urlunsplit()` (dans le module `urllib.parse`), 1124
- `urn` (attribut `uuid.UUID`), 1171
- `use_default_colors()` (dans le module `curses`), 657
- `use_env()` (dans le module `curses`), 656
- `use_rawinput` (attribut `cmd.Cmd`), 1295
- `UseForeignDTD()` (méthode `xml.parsers.expat.xmlparser`), 1075
- `USER`, 650
- `user`
 - `effective id`, 522
 - `id`, 523
 - `id`, setting, 525
- `user()` (méthode `poplib.POP3`), 1144
- `USER_BASE` (dans le module `site`), 1607
- `user_call()` (méthode `bdb.Bdb`), 1485
- `user_exception()` (méthode `bdb.Bdb`), 1485
- `user_line()` (méthode `bdb.Bdb`), 1485
- `user_return()` (méthode `bdb.Bdb`), 1485
- `USER_SITE` (dans le module `site`), 1607
- `--user-base`
 - site command line option, 1608
- `usercustomize`
 - module, 1606
- `UserDict` (classe dans `collections`), 217
- `UserList` (classe dans `collections`), 217
- `USERNAME`, 522, 650
- `username` (attribut `email.headerregistry.Address`), 957
- `USERPROFILE`, 369
- `userptr()` (méthode `curses.panel.Panel`), 672
- `--user-site`
 - site command line option, 1608
- `UserString` (classe dans `collections`), 217
- `UserWarning`, 88
- `USTAR_FORMAT` (dans le module `tarfile`), 462
- `UTC`, 574
- `utc` (attribut `datetime.timezone`), 195
- `utcfromtimestamp()` (méthode de la classe `datetime.datetime`), 178
- `utcnow()` (méthode de la classe `datetime.datetime`), 177
- `utcoffset()` (méthode `datetime.datetime`), 181
- `utcoffset()` (méthode `datetime.time`), 187
- `utcoffset()` (méthode `datetime.timezone`), 195
- `utcoffset()` (méthode `datetime.tzinfo`), 188
- `utctimetuple()` (méthode `datetime.datetime`), 181
- `utf8` (attribut `email.policy.EmailPolicy`), 949
- `utf8()` (méthode `poplib.POP3`), 1145
- `utf8_enabled` (attribut `imaplib.IMAP4`), 1152
- `utime()` (dans le module `os`), 547
- `uu`
 - module, 1019
- `uu` (module), 1021
- `UUID` (classe dans `uuid`), 1170
- `uuid` (module), 1170
- `uuid1`, 1172
- `uuid1()` (dans le module `uuid`), 1172
- `uuid3`, 1172
- `uuid3()` (dans le module `uuid`), 1172
- `uuid4`, 1172
- `uuid4()` (dans le module `uuid`), 1172
- `uuid5`, 1172
- `uuid5()` (dans le module `uuid`), 1172
- `UuidCreate()` (dans le module `msilib`), 1683
- ## V
- `-v`
 - `tarfile` command line option, 468
 - `timeit` command line option, 1506

- unittest-discover command line option, 1389
- v4_int_to_packed() (dans le module *ipaddress*), 1221
- v6_int_to_packed() (dans le module *ipaddress*), 1221
- validator() (dans le module *wsgiref.validate*), 1100
- value
 - truth, 27
- value (attribut *ctypes._SimpleCData*), 709
- value (attribut *http.cookiejar.Cookie*), 1196
- value (attribut *http.cookies.Morsel*), 1188
- value (attribut *xml.dom.Attr*), 1052
- Value() (dans le module *multiprocessing*), 743
- Value() (dans le module *multiprocessing.sharedctypes*), 745
- Value() (méthode *multiprocessing.managers.SyncManager*), 748
- value_decode() (méthode *http.cookies.BaseCookie*), 1187
- value_encode() (méthode *http.cookies.BaseCookie*), 1187
- ValueError, 87
- valuerefs() (méthode *weakref.WeakValueDictionary*), 233
- values
 - Boolean, 79
- values() (méthode *dict*), 74
- values() (méthode *email.message.EmailMessage*), 934
- values() (méthode *email.message.Message*), 970
- values() (méthode *mailbox.Mailbox*), 997
- values() (méthode *types.MappingProxyType*), 241
- ValuesView (classe dans *collections.abc*), 220
- ValuesView (classe dans *typing*), 1356
- variable de classe, 1755
- variable d'environnement
 - AUDIODEV, 1238
 - BROWSER, 1085, 1086
 - COLS, 656
 - COLUMNS, 657
 - COMSPEC, 555, 778
 - HOME, 369
 - HOMEDRIVE, 369
 - HOMEPAH, 369
 - http_proxy, 1105, 1118
 - IDLE*STARTUP, 1342
 - KDEDIR, 1087
 - LANG, 1244, 1245, 1252, 1255
 - LANGUAGE, 1244, 1245
 - LC_ALL, 1244, 1245
 - LC_MESSAGES, 1244, 1245
 - LINES, 652, 656, 657
 - LNAME, 650
 - LOGNAME, 522, 650
 - MIXERDEV, 1238
 - no_proxy, 1107
 - PAGER, 1363
 - PATH, 551, 554, 561, 1085, 1093, 1094
 - POSIXLY_CORRECT, 612
 - PYTHON_DOM, 1046
 - PYTHONASYNCIODEBUG, 866, 909
 - PYTHONDOCS, 1363
 - PYTHONDONTWRITEBYTECODE, 1541
 - PYTHONFAULTHANDLER, 1487
 - PYTHONIOENCODING, 1552
 - PYTHONLEGACYWINDOWSFSENCODING, 1552
 - PYTHONNOUSERSITE, 1607
 - PYTHONPATH, 1093, 1548
 - PYTHONSTARTUP, 142, 1342, 1547, 1607
 - PYTHONTRACEMALLOC, 1511, 1515
 - PYTHONUSERBASE, 1607
 - SSL_CERT_FILE, 846
 - SSL_CERT_PATH, 846
 - SystemRoot, 779
 - TEMP, 383
 - TERM, 655, 656
 - TIX_LIBRARY, 1331
 - TMP, 383
 - TMPDIR, 383
 - TZ, 579, 580
 - USER, 650
 - USERNAME, 522, 650
 - USERPROFILE, 369
- variance() (dans le module *statistics*), 319
- variant (attribut *uuid.UUID*), 1171
- vars() (fonction de base), 22
- vbar (attribut *tkinter.scrolledtext.ScrolledText*), 1336
- VBAR (dans le module *token*), 1651
- VBAREQUAL (dans le module *token*), 1651
- Vec2D (classe dans *turtle*), 1288
- venv (module), 1523
- verbose
 - tarfile command line option, 468
 - timeit command line option, 1506
 - unittest-discover command line option, 1389
- VERBOSE (dans le module *re*), 109
- verbose (dans le module *tabnanny*), 1657
- verbose (dans le module *test.support*), 1476
- verify() (méthode *smtplib.SMTP*), 1161
- verify_client_post_handshake() (méthode *ssl.SSLSocket*), 831
- VERIFY_CRL_CHECK_CHAIN (dans le module *ssl*), 824
- VERIFY_CRL_CHECK_LEAF (dans le module *ssl*), 824
- VERIFY_DEFAULT (dans le module *ssl*), 824
- verify_flags (attribut *ssl.SSLContext*), 837
- verify_mode (attribut *ssl.SSLContext*), 837

- ul style="list-style-type: none; padding-left: 0;">
- `verify_request()` (méthode `socketserver.BaseServer`), 1177
- `VERIFY_X509_STRICT` (dans le module `ssl`), 824
- `VERIFY_X509_TRUSTED_FIRST` (dans le module `ssl`), 824
- `VerifyFlags` (classe dans `ssl`), 824
- `VerifyMode` (classe dans `ssl`), 824
- verrou global de l'interpréteur, 1758
- `--version`
 - trace command line option, 1508
- `version` (attribut `email.headerregistry.MIMEVersionHeader`), 955
- `version` (attribut `http.client.HTTPResponse`), 1136
- `version` (attribut `http.cookiejar.Cookie`), 1196
- `version` (attribut `ipaddress.IPv4Address`), 1212
- `version` (attribut `ipaddress.IPv4Network`), 1216
- `version` (attribut `ipaddress.IPv6Address`), 1213
- `version` (attribut `ipaddress.IPv6Network`), 1218
- `version` (attribut `urllib.request.URLopener`), 1120
- `version` (attribut `uuid.UUID`), 1171
- `version` (dans le module `curses`), 663
- `version` (dans le module `marshal`), 414
- `version` (dans le module `sqlite3`), 420
- `version` (dans le module `sys`), 1553
- `version()` (dans le module `ensurepip`), 1523
- `version()` (dans le module `platform`), 673
- `version()` (méthode `ssl.SSLSocket`), 831
- `version_info` (dans le module `sqlite3`), 420
- `version_info` (dans le module `sys`), 1553
- `version_string()` (méthode `http.server.BaseHTTPRequestHandler`), 1184
- `vformat()` (méthode `string.Formatter`), 92
- `virtual`
 - Environments, 1523
- `visit()` (méthode `ast.NodeVisitor`), 1648
- `vline()` (méthode `curses.window`), 663
- `voidcmd()` (méthode `ftplib.FTP`), 1141
- `volume` (attribut `zipfile.ZipInfo`), 459
- `vonmisesvariate()` (dans le module `random`), 312
- vue de dictionnaire, 1756
- ## W
- `W_OK` (dans le module `os`), 534
 - `wait()` (dans le module `asyncio`), 881
 - `wait()` (dans le module `concurrent.futures`), 773
 - `wait()` (dans le module `multiprocessing.connection`), 756
 - `wait()` (dans le module `os`), 556
 - `wait()` (méthode `asyncio.asyncio.subprocess.Process`), 900
 - `wait()` (méthode `asyncio.Condition`), 906
 - `wait()` (méthode `asyncio.Event`), 905
 - `wait()` (méthode `multiprocessing.pool.AsyncResult`), 754
 - `wait()` (méthode `subprocess.Popen`), 780
 - `wait()` (méthode `threading.Barrier`), 725
 - `wait()` (méthode `threading.Condition`), 722
 - `wait()` (méthode `threading.Event`), 724
 - `wait3()` (dans le module `os`), 557
 - `wait4()` (dans le module `os`), 557
 - `wait_closed()` (méthode `asyncio.Server`), 867
 - `wait_for()` (dans le module `asyncio`), 882
 - `wait_for()` (méthode `asyncio.Condition`), 906
 - `wait_for()` (méthode `threading.Condition`), 722
 - `waitid()` (dans le module `os`), 556
 - `waitpid()` (dans le module `os`), 557
 - `walk()` (dans le module `ast`), 1647
 - `walk()` (dans le module `os`), 547
 - `walk()` (méthode `email.message.EmailMessage`), 937
 - `walk()` (méthode `email.message.Message`), 973
 - `walk_packages()` (dans le module `pkgutil`), 1618
 - `walk_stack()` (dans le module `traceback`), 1582
 - `walk_tb()` (dans le module `traceback`), 1582
 - `want` (attribut `doctest.Example`), 1379
 - `warn()` (dans le module `warnings`), 1562
 - `warn_explicit()` (dans le module `warnings`), 1562
 - `Warning`, 88, 431
 - `warning()` (dans le module `logging`), 625
 - `warning()` (méthode `logging.Logger`), 617
 - `warning()` (méthode `xml.sax.handler.ErrorHandler`), 1068
 - `warnings`, 1558
 - `warnings` (module), 1558
 - `WarningsRecorder` (classe dans `test.support`), 1481
 - `warnoptions` (dans le module `sys`), 1553
 - `wasSuccessful()` (méthode `unittest.TestResult`), 1408
 - `WatchedFileHandler` (classe dans `logging.handlers`), 640
 - `wave` (module), 1231
 - `WCONTINUED` (dans le module `os`), 557
 - `WCOREDUMP()` (dans le module `os`), 558
 - `WeakKeyDictionary` (classe dans `weakref`), 233
 - `WeakMethod` (classe dans `weakref`), 234
 - `weakref` (module), 231
 - `WeakSet` (classe dans `weakref`), 233
 - `WeakValueDictionary` (classe dans `weakref`), 233
 - `webbrowser` (module), 1085
 - `weekday()` (dans le module `calendar`), 201
 - `weekday()` (méthode `datetime.date`), 175
 - `weekday()` (méthode `datetime.datetime`), 182
 - `weekheader()` (dans le module `calendar`), 201
 - `weibullvariate()` (dans le module `random`), 312
 - `WEXITED` (dans le module `os`), 556
 - `WEXITSTATUS()` (dans le module `os`), 558
 - `wfile` (attribut `http.server.BaseHTTPRequestHandler`), 1182
 - `what()` (dans le module `imghdr`), 1236
 - `what()` (dans le module `sndhdr`), 1237
 - `whathdr()` (dans le module `sndhdr`), 1237
 - `whatis` (`pdb` command), 1494

- where (*pdb* command), 1492
- which() (*dans le module shutil*), 391
- whichdb() (*dans le module dbm*), 415
- while
 - état, 27
- whitespace (*attribut shlex.shlex*), 1300
- whitespace (*dans le module string*), 92
- whitespace_split (*attribut shlex.shlex*), 1300
- Widget (*classe dans tkinter.ttk*), 1318
- width (*attribut textwrap.TextWrapper*), 134
- width() (*dans le module turtle*), 1272
- WIFCONTINUED() (*dans le module os*), 558
- WIFEXITED() (*dans le module os*), 558
- WIFSIGNALED() (*dans le module os*), 558
- WIFSTOPPED() (*dans le module os*), 558
- win32_ver() (*dans le module platform*), 674
- WinDLL (*classe dans ctypes*), 701
- window manager (*widgets*), 1311
- window() (*méthode curses.panel.Panel*), 672
- window_height() (*dans le module turtle*), 1286
- window_width() (*dans le module turtle*), 1287
- Windows ini file, 478
- WindowsError, 87
- WindowsPath (*classe dans pathlib*), 361
- WindowsRegistryFinder (*classe dans importlib.machinery*), 1630
- winerror (*attribut OSError*), 84
- WinError() (*dans le module ctypes*), 708
- WINFUNCTYPE() (*dans le module ctypes*), 704
- winreg (*module*), 1690
- WinSock, 848
- winsound (*module*), 1698
- winver (*dans le module sys*), 1553
- WITH_CLEANUP_FINISH (*opcode*), 1670
- WITH_CLEANUP_START (*opcode*), 1669
- with_hostmask (*attribut ipaddress.IPv4Interface*), 1220
- with_hostmask (*attribut ipaddress.IPv4Network*), 1216
- with_hostmask (*attribut ipaddress.IPv6Interface*), 1220
- with_hostmask (*attribut ipaddress.IPv6Network*), 1218
- with_name() (*méthode pathlib.PurePath*), 360
- with_netmask (*attribut ipaddress.IPv4Interface*), 1220
- with_netmask (*attribut ipaddress.IPv4Network*), 1216
- with_netmask (*attribut ipaddress.IPv6Interface*), 1220
- with_netmask (*attribut ipaddress.IPv6Network*), 1218
- with_prefixlen (*attribut ipaddress.IPv4Interface*), 1220
- with_prefixlen (*attribut ipaddress.IPv4Network*), 1216
- with_prefixlen (*attribut ipaddress.IPv6Interface*), 1220
- with_prefixlen (*attribut ipaddress.IPv6Network*), 1218
- with_suffix() (*méthode pathlib.PurePath*), 360
- with_traceback() (*méthode BaseException*), 82
- WNOHANG (*dans le module os*), 557
- WNOWAIT (*dans le module os*), 556
- wordchars (*attribut shlex.shlex*), 1300
- World Wide Web, 1085, 1122, 1129
- wrap() (*dans le module textwrap*), 132
- wrap() (*méthode textwrap.TextWrapper*), 135
- wrap_bio() (*méthode ssl.SSLContext*), 836
- wrap_future() (*dans le module asyncio*), 880
- wrap_socket() (*dans le module ssl*), 819
- wrap_socket() (*méthode ssl.SSLContext*), 835
- wrapper() (*dans le module curses*), 657
- wraps() (*dans le module functools*), 341
- WRITABLE (*dans le module tkinter*), 1314
- writable() (*méthode asyncore.dispatcher*), 916
- writable() (*méthode io.IOBase*), 566
- write() (*dans le module os*), 532
- write() (*dans le module turtle*), 1275
- write() (*méthode asyncio.StreamWriter*), 894
- write() (*méthode asyncio.WriteTransport*), 884
- write() (*méthode codecs.StreamWriter*), 157
- write() (*méthode code.InteractiveInterpreter*), 1612
- write() (*méthode configparser.ConfigParser*), 492
- write() (*méthode email.generator.BytesGenerator*), 944
- write() (*méthode email.generator.Generator*), 945
- write() (*méthode io.BufferedIOBase*), 568
- write() (*méthode io.BufferedWriter*), 570
- write() (*méthode io.RawIOBase*), 567
- write() (*méthode io.TextIOBase*), 571
- write() (*méthode mmap.mmap*), 929
- write() (*méthode ossaudiodev.oss_audio_device*), 1238
- write() (*méthode ssl.MemoryBIO*), 844
- write() (*méthode ssl.SSLSocket*), 829
- write() (*méthode telnetlib.Telnet*), 1169
- write() (*méthode xml.etree.ElementTree.ElementTree*), 1041
- write() (*méthode zipfile.ZipFile*), 457
- write_byte() (*méthode mmap.mmap*), 929
- write_bytes() (*méthode pathlib.Path*), 367
- write_docstringdict() (*dans le module turtle*), 1290
- write_eof() (*méthode asyncio.StreamWriter*), 894
- write_eof() (*méthode asyncio.WriteTransport*), 885
- write_eof() (*méthode ssl.MemoryBIO*), 844
- write_history_file() (*dans le module readline*), 140
- write_results() (*méthode trace.CoverageResults*), 1510
- write_text() (*méthode pathlib.Path*), 367
- writeall() (*méthode ossaudiodev.oss_audio_device*), 1238
- writeframes() (*méthode aifc.aifc*), 1228
- writeframes() (*méthode sunau.AU_write*), 1231

[writeframes\(\) \(méthode wave.Wave_write\), 1233](#)
[writeframesraw\(\) \(méthode aifc.aifc\), 1228](#)
[writeframesraw\(\) \(méthode sunau.AU_write\), 1231](#)
[writeframesraw\(\) \(méthode wave.Wave_write\), 1233](#)
[writeheader\(\) \(méthode csv.DictWriter\), 476](#)
[writelines\(\) \(méthode asyncio.StreamWriter\), 894](#)
[writelines\(\) \(méthode asyncio.WriteTransport\), 884](#)
[writelines\(\) \(méthode codecs.StreamWriter\), 158](#)
[writelines\(\) \(méthode io.IOBase\), 566](#)
[writePlist\(\) \(dans le module plistlib\), 500](#)
[writePlistToBytes\(\) \(dans le module plistlib\), 500](#)
[writepy\(\) \(méthode zipfile.PyZipFile\), 458](#)
[writer \(attribut formatter.formatter\), 1678](#)
[writer\(\) \(dans le module csv\), 472](#)
[writerow\(\) \(méthode csv.csvwriter\), 476](#)
[writerows\(\) \(méthode csv.csvwriter\), 476](#)
[writestr\(\) \(méthode zipfile.ZipFile\), 457](#)
[WriteTransport \(classe dans asyncio\), 884](#)
[writev\(\) \(dans le module os\), 532](#)
[writexml\(\) \(méthode xml.dom.minidom.Node\), 1057](#)
[WrongDocumentErr, 1054](#)
[ws_comma \(2to3 fixer\), 1473](#)
[wsgi_file_wrapper \(attribut ref.handlers.BaseHandler\), 1103](#)
[wsgi_multiprocess \(attribut ref.handlers.BaseHandler\), 1102](#)
[wsgi_multithread \(attribut ref.handlers.BaseHandler\), 1102](#)
[wsgi_run_once \(attribut ref.handlers.BaseHandler\), 1102](#)
[wsgiref \(module\), 1095](#)
[wsgiref.handlers \(module\), 1101](#)
[wsgiref.headers \(module\), 1098](#)
[wsgiref.simple_server \(module\), 1099](#)
[wsgiref.util \(module\), 1096](#)
[wsgiref.validate \(module\), 1100](#)
[WSGIRequestHandler \(classe dans wsgiref.simple_server\), 1099](#)
[WSGIServer \(classe dans wsgiref.simple_server\), 1099](#)
[wShowWindow \(attribut subprocess.STARTUPINFO\), 782](#)
[WSTOPPED \(dans le module os\), 556](#)
[WSTOPSIG\(\) \(dans le module os\), 558](#)
[wstring_at\(\) \(dans le module ctypes\), 708](#)
[WTERMSIG\(\) \(dans le module os\), 558](#)
[WUNTRACED \(dans le module os\), 557](#)
[WWW, 1085, 1122, 1129](#)
[server, 1088, 1181](#)

X

[X \(dans le module re\), 109](#)
[-x regex](#)
[compileall command line option, 1660](#)
[X509 certificate, 837](#)

[X_OK \(dans le module os\), 534](#)
[xatom\(\) \(méthode imaplib.IMAP4\), 1151](#)
[XATTR_CREATE \(dans le module os\), 550](#)
[XATTR_REPLACE \(dans le module os\), 550](#)
[XATTR_SIZE_MAX \(dans le module os\), 550](#)
[xcor\(\) \(dans le module turtle\), 1270](#)
[XDR, 496](#)
[xdrlib \(module\), 496](#)
[xhdr\(\) \(méthode nntplib.NNTP\), 1158](#)
[XHTML, 1024](#)
[XHTML_NAMESPACE \(dans le module xml.dom\), 1046](#)
[xml \(module\), 1029](#)
[XML\(\) \(dans le module xml.etree.ElementTree\), 1039](#)
[XML_ERROR_ABORTED \(dans le module xml.parsers.expat.errors\), 1082](#)
[XML_ERROR_ASYNC_ENTITY \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_BAD_CHAR_REF \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_BINARY_ENTITY_REF \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING \(dans le module xml.parsers.expat.errors\), 1082](#)
[XML_ERROR_DUPLICATE_ATTRIBUTE \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_ENTITY_DECLARED_IN_PE \(dans le module xml.parsers.expat.errors\), 1082](#)
[XML_ERROR_EXTERNAL_ENTITY_HANDLING \(dans le module xml.parsers.expat.errors\), 1082](#)
[XML_ERROR_FEATURE_REQUIRES_XML_DTD \(dans le module xml.parsers.expat.errors\), 1082](#)
[XML_ERROR_FINISHED \(dans le module xml.parsers.expat.errors\), 1082](#)
[XML_ERROR_INCOMPLETE_PE \(dans le module xml.parsers.expat.errors\), 1082](#)
[XML_ERROR_INCORRECT_ENCODING \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_INVALID_TOKEN \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_JUNK_AFTER_DOC_ELEMENT \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_MISPLACED_XML_PI \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_NO_ELEMENTS \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_NO_MEMORY \(dans le module xml.parsers.expat.errors\), 1081](#)
[XML_ERROR_NOT_STANDALONE \(dans le module xml.parsers.expat.errors\), 1082](#)
[XML_ERROR_NOT_SUSPENDED \(dans le module xml.parsers.expat.errors\), 1082](#)
[XML_ERROR_PARAM_ENTITY_REF \(dans le module](#)

- `xml.parsers.expat.errors`), 1081
 - `XML_ERROR_PARTIAL_CHAR` (dans le module `xml.parsers.expat.errors`), 1081
 - `XML_ERROR_PUBLICID` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_RECURSIVE_ENTITY_REF` (dans le module `xml.parsers.expat.errors`), 1081
 - `XML_ERROR_SUSPEND_PE` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_SUSPENDED` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_SYNTAX` (dans le module `xml.parsers.expat.errors`), 1081
 - `XML_ERROR_TAG_MISMATCH` (dans le module `xml.parsers.expat.errors`), 1081
 - `XML_ERROR_TEXT_DECL` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_UNBOUND_PREFIX` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_UNCLOSED_CDATA_SECTION` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_UNCLOSED_TOKEN` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_UNDECLARING_PREFIX` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_UNDEFINED_ENTITY` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_UNEXPECTED_STATE` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_UNKNOWN_ENCODING` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_ERROR_XML_DECL` (dans le module `xml.parsers.expat.errors`), 1082
 - `XML_NAMESPACE` (dans le module `xml.dom`), 1046
 - `xmlcharrefreplace_errors()` (dans le module `codecs`), 155
 - `XmlDeclHandler()` (méthode `xml.parsers.expat.xmlparser`), 1077
 - `xml.dom` (module), 1045
 - `xml.dom.minidom` (module), 1056
 - `xml.dom.pulldom` (module), 1060
 - `xml.etree.ElementTree` (module), 1030
 - `XMLFilterBase` (classe dans `xml.sax.saxutils`), 1069
 - `XMLGenerator` (classe dans `xml.sax.saxutils`), 1069
 - `XMLID()` (dans le module `xml.etree.ElementTree`), 1039
 - `XMLNS_NAMESPACE` (dans le module `xml.dom`), 1046
 - `XMLParser` (classe dans `xml.etree.ElementTree`), 1043
 - `xml.parsers.expat` (module), 1074
 - `xml.parsers.expat.errors` (module), 1081
 - `xml.parsers.expat.model` (module), 1080
 - `XMLParserType` (dans le module `xml.parsers.expat`), 1074
 - `XMLPullParser` (classe dans `xml.etree.ElementTree`), 1044
 - `XMLReader` (classe dans `xml.sax.xmlreader`), 1070
 - `xmlrpc.client` (module), 1198
 - `xmlrpc.server` (module), 1205
 - `xml.sax` (module), 1062
 - `xml.sax.handler` (module), 1064
 - `xml.sax.saxutils` (module), 1069
 - `xml.sax.xmlreader` (module), 1070
 - `xor()` (dans le module `operator`), 344
 - `xover()` (méthode `nntplib.NNTP`), 1158
 - `xpath()` (méthode `nntplib.NNTP`), 1158
 - `xrange` (2to3 fixer), 1473
 - `xreadlines` (2to3 fixer), 1473
 - `xview()` (méthode `tkinter.ttk.Treeview`), 1327
- ## Y
- `Y2K`, 573
 - `ycor()` (dans le module `turtle`), 1270
 - `year` (attribut `datetime.date`), 174
 - `year` (attribut `datetime.datetime`), 179
 - `Year 2000`, 573
 - `Year 2038`, 573
 - `yeardatescalendar()` (méthode `calendar.Calendar`), 199
 - `yeardays2calendar()` (méthode `calendar.Calendar`), 199
 - `yeardayscalendar()` (méthode `calendar.Calendar`), 200
 - `YESEXPR` (dans le module locale), 1254
 - `YIELD_FROM` (opcode), 1669
 - `YIELD_VALUE` (opcode), 1669
 - `yiq_to_rgb()` (dans le module `colorsys`), 1235
 - `yview()` (méthode `tkinter.ttk.Treeview`), 1327
- ## Z
- `ZeroDivisionError`, 87
 - `zfill()` (méthode `bytearray`), 62
 - `zfill()` (méthode `bytes`), 62
 - `zfill()` (méthode `str`), 49
 - `zip` (2to3 fixer), 1473
 - `zip()` (fonction de base), 22
 - `ZIP_BZIP2` (dans le module `zipfile`), 454
 - `ZIP_DEFLATED` (dans le module `zipfile`), 454
 - `zip_longest()` (dans le module `itertools`), 331
 - `ZIP_LZMA` (dans le module `zipfile`), 454
 - `ZIP_STORED` (dans le module `zipfile`), 454
 - `zipapp` (module), 1531
 - `zipapp` command line option
 - `-h`, 1532
 - `--help`, 1532
 - `--info`, 1532
 - `-m <mainfn>`, 1532
 - `--main=<mainfn>`, 1532
 - `-o <output>`, 1532
 - `--output=<output>`, 1532

- p <interpreter>, 1532
 - python=<interpreter>, 1532
- ZipFile (*classe dans zipfile*), 454
- zipfile (*module*), 453
- zipfile command line option
 - c <zipfile> <source1> ...
 <sourceN>, 460
 - e <zipfile> <output_dir>, 460
 - l <zipfile>, 460
 - t <zipfile>, 460
- zipimport (*module*), 1615
- zipimporter (*classe dans zipimport*), 1616
- ZipImportError, 1616
- ZipInfo (*classe dans zipfile*), 454
- zlib (*module*), 439
- ZLIB_RUNTIME_VERSION (*dans le module zlib*), 442
- ZLIB_VERSION (*dans le module zlib*), 442